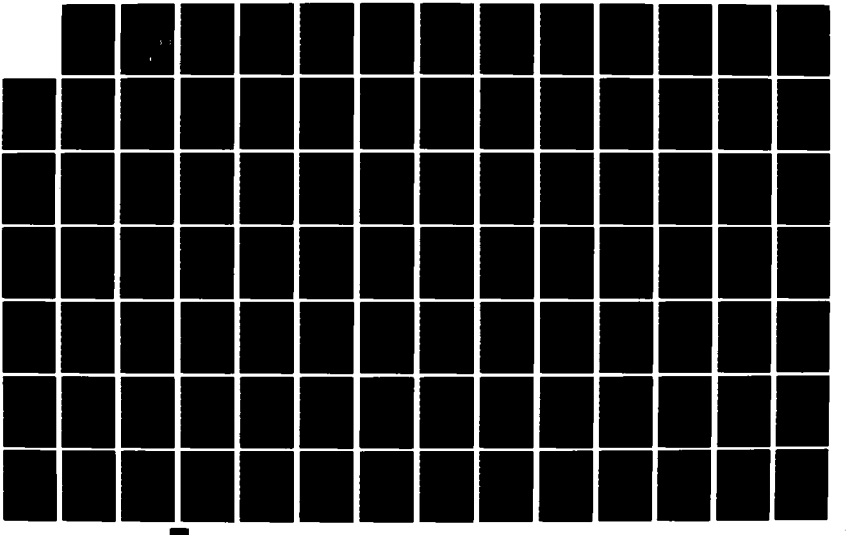
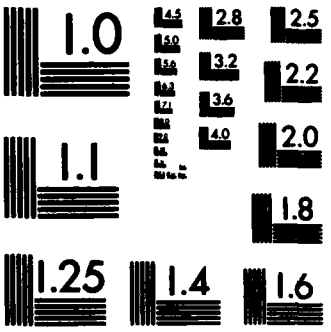
AD-A134 092 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR ADA INTEGRATED ENVIRONMENT.. (U) INTERMETRICS INC CAMBRIDGE MA 12 NOV 82 IR-678-2 F30602-80-C-0291 1/2

UNCLASSIFIED F/G 9/2 NL





CONTRACT F30602-80-C-0291

IR-678-2
COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR Ada INTEGRATED ENVIRONMENT: KAPSE/DATABASE TYPE B5 B5-AIE (1).KAPSE (1)

12 NOVEMBER 1982

PREPARED FOR: ROME AIR DEVELOPMENT CENTER
CONTRACTING DIVISION/PKRD
GRIFFISS AFB, N.Y. 13441

PREPARED BY: INTERMETRICS, INC.
733 CONCORD AVE.
CAMBRIDGE, MA 02138

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Donald Mark is the Program Engineer for the Air Force. Mr. Mike Ryer is the Project Manager for Intermetrics.

CONTENTS

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

FIGURES

1. SCOPE

1.1 Identification

This specification establishes the requirements for performance, design, test, and qualification of a set of computer program modules identified as the Kernel Ada Programming Support Environment (KAPSE) of the Ada Integrated Environment (AIE).

1.2 Functional Summary

The KAPSE provides several facilities to the Ada Programming Support Environment (APSE), which can be grouped into the following five Computer Program Configuration Items (CPCIs):

1) SIMPCOMP — Database Operations on Simple and Composite Objects.

2) ACCECAT — Access Control and Categorization of Database Objects, and the Manipulation of User-Defined Attributes.

3) MULTPROG — Invocation of and Communication Between Multiple Ada Programs, plus Multi-User and Multi-KAPSE Support and Synchronization.

4) HISTARCH — Configuration and System Management, with History, Archiving, Backup, and Recovery.

5) RTS — Run-Time Support for the Execution of Ada Programs, including Language-Defined Input/Output Packages.

This specification identifies the functional capabilities of the various KAPSE computer program components (CPCs) and describes the KAPSE/tool interfaces as well as the KAPSE/Host computer interfaces.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

This page left blank intentionally.

2. APPLICABLE DOCUMENTS

Note that bracketed or inter-subsystem identifiers are used to refer to documents in the text.

2.1 Program Definition Documents

[STONEMAN80] Requirements for Ada Programming Support Environments, "STONEMAN," Department of Defense, February 1980.

[SOW80] Revised Statement of Work, 15 March 1980.

[LRM] Reference Manual for the Ada Programming Language, Draft Standard Document, U.S. Department of Defense, July 1982.

2.2 Inter-Subsystem Specifications

System Specification for Ada Integrated Environment, AIE(1).

Computer Program Development Specifications for Ada Integrated Environment (Type B5):

a. Ada Compiler Phases, AIE(1).COMP(1).

b. MAPSE Command Processor, AIE(1).MCP(1).

c. MAPSE Generation and Support, AIE(1).MGS(1).

d. Program Integration Facilities, AIE(1).PIF(1).

e. MAPSE Debugging Facilities, AIE(1).DBUG(1).

f. MAPSE Text Editor, AIE(1).TXED(1).

g. Virtual Memory Methodology, AIE(1).VMM(2).

h. Technical Report (Interim) IR-684.

2.3 Military Specifications and Standards

Data Item Description DI-E-30139, USAF, 14 July 1976.

2.4 Miscellaneous Documents

[IBM81] IBM Virtual Machine/System Product: System Programmer's Guide, SC19-6203-0, International Business Machines, Inc., December 1981.

[PE79] OS/32 Programmer Reference Manual, Perkin-Elmer Computer Systems Division, Oceanport, NJ, April 1979.

[Knuth73] The Art of Computer Programming, V. 3., Donald Knuth, Addison Wesley, 1973.

[Warshall80] "A Theory of Accountability," Stephen Warshall, CADD-8011-2401, Mass. Computer Assoc., Inc., Wakefield, MA, November 1980.

3. REQUIREMENTS

3.1 Introduction

This section provides the set of requirements for the KAPSE of the Ada Integrated Environment. This includes the performance and interface specifications to which the KAPSE must comply.

3.1.1 General Description

The KAPSE provides database, program invocation, and run-time support for all MAPSE tools and user Ada programs. In so far as possible, the KAPSE isolates the rest of the AIE from host machine idiosyncrasies, making the entire MAPSE toolset and user-developed programs easily portable from one AIE to another. Figure 3-1 is an overview of the KAPSE and its interfaces.

3.1.2 Peripheral Equipment Identification

The 4341.KAPSE shall interface with the following equipment on an IBM 4341 computer system:

a. 4Mb 4341 Central Processing Unit, Group I;

b. 3410 Magnetic Tape Drive;

c. 3705 Communications Controller;

d. 3278 Half-Duplex Full-Screen Display Terminals;

e. 9712 8-line ASCII 1200 baud Half-Duplex TTY terminal controller;

f. 3375 Direct Access Storage Devices (4 drives, with 600Mb each)

g. 3203 High Speed Line Printer;

The 832.KAPSE shall interface with the following equipment on a Perkin-Elmer (PE) 8/32 computer system:

a. 8/32 Central Processing Unit, with <<TBD>> memory;

b. <<TBD>> Magnetic Tape Drive;

c. <<TBD>> Full-Duplex ASCII Terminals;

d. <<TBD>> Disk;

e. <<TBD>> Line Printer;

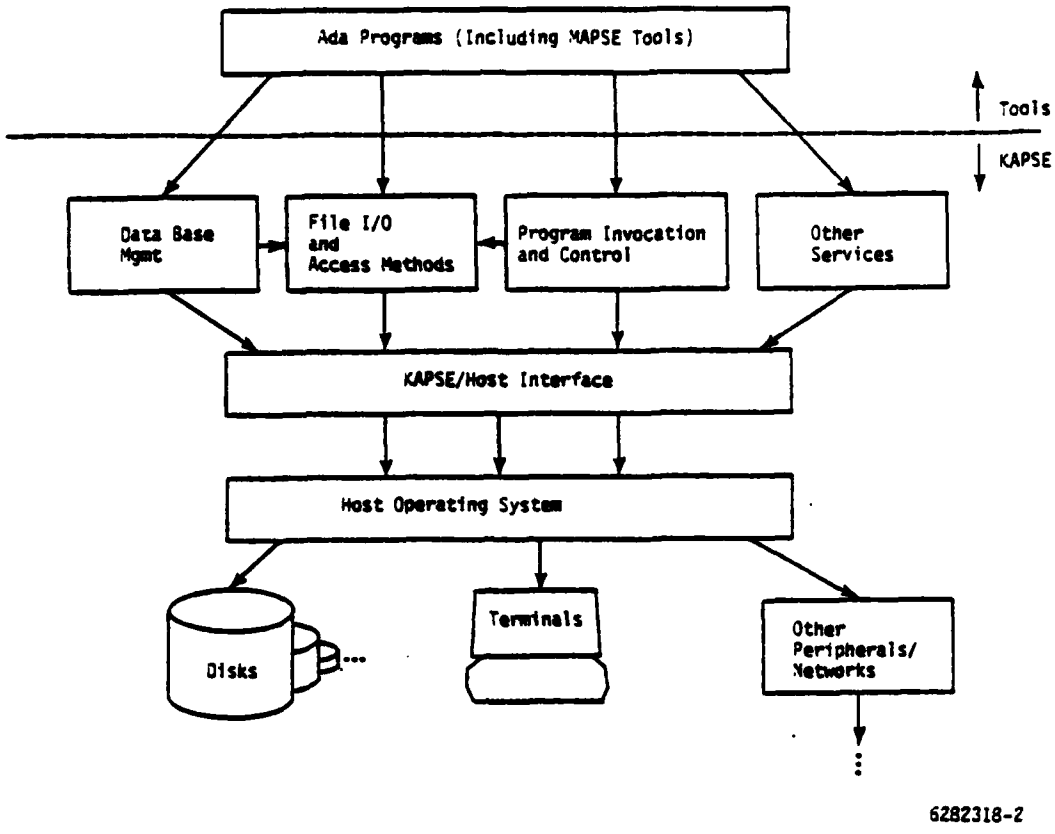


Figure 3-1: KAPSE/Database Overview and Computer Interface Diagram

3.1.3 Interface Identification

The KAPSE shall interface with all subsystems within the AIE, including:

a. MCP — MAPSE Command Processor;

b. COMP — Ada Compiler;

c. TXED — Text Editor;

d. DBUG — MAPSE Debugger;

e. PIF — Program Integration Facility;

f. VMM Virtual Memory Methodology;

g. MGS MAPSE Generation and Support;

In every case, the KAPSE is providing the interface to the program. The programs are all users of the KAPSE/Tool interface (see 3.2.4.1) and the Database/Tool interface (see 3.2.4.3). COMP uses the Compiler/Run-Time System interface (see 3.2.4.5). PIF uses the Linker/Loader interface (see 3.2.4.6).

The 4341.KAPSE interfaces with VM/SP [IBM81]. The KAPSE is the user of this interface, VM/SP is the provider of the interface. All uses of this interface are encapsulated within the KAPSE/Host interface packages.

The 832.KAPSE interfaces with OS 8/32 [PE79]. The KAPSE is the user of this interface, OS 8/32 is the provider of the interface. All uses of this interface are encapsulated within the KAPSE/Host interface packages.

3.2 Functional Description

3.2.1 Equipment Descriptions

The following equipment of the 4341 computer system impose requirements on components of the KAPSE:

a. 4Mb 4341 Central Processing Unit, Group I;

The run-time system (KAPSE.RTS) must be written to be consistent with the interrupt and time clock facilities of the 4341. In particular, interrupt identification information must be fetched from the appropriate fixed locations in low memory of the virtual machine address space upon interrupt. The Set Clock Comparator instruction must be used to handle the real-time-oriented tasking constructs.

The real memory capacity of the 4341 places no direct requirements on the KAPSE because it runs in a virtual machine provided by VM/SP. The addressing limitations of the machine (16Mb) do represent an upper bound on the size of a single virtual machine, and hence on the number of programs which can simultaneously reside in the virtual memory space. This bound imposes limitations on the PROGRAM_LOADING package of the KAPSE/Host interface (KAPSE.MULTPROG).

The instruction set of the 4341 processor places requirements on the efficient design of the run-time system routines (KAPSE.RTS), in particular in the CPC of unit execution support.

b. 3410 Magnetic Tape Drive;

The <<TBD>> tape device driver (KAPSE.SIMPCOMP.DEVICE_IO) must be consistent with the control and status register layouts of the 3411 tape controller.

c. 3705 Communications Controller;

The terminal driver (KAPSE.SIMPCOMP.DEVICE_IO) must be consistent with the facilities of the 3705 front-end. In particular, the half-duplex nature of the 3705 must be accommodated when implementing the host-independent DEVICE_IO interface.

d. 3278 Half-Duplex Full-Screen Display Terminals;

The terminal driver (KAPSE.SIMPCOMP.DEVICE_IO) must perform EBCDIC to ASCII conversions for 3278 terminals, as well as accommodate the field-oriented nature of the 3278 while

providing a random-access host-independent interface via the package TERMINAL_IO.

e. 9712 8-line ASCII 1200 baud Half-Duplex TTY terminal controller;

The terminal driver (KAPSE.SIMPCOMP.DEVICE_IO) must convert random-access cursor positioning requests to the proper control character sequences for the connected ASCII terminals.

f. 3375 Direct Access Storage Device

The disk device channel driver (KAPSE.SIMPCOMP.BLOCK_IO) must correctly set up channel programs to perform the fixed-block-size read and write requests as part of implementing the host-independent interface of the package PHYS_BLOCK_IO.

g. 3203 High Speed Line Printer;

The line printer device driver (KAPSE.SIMPCOMP.DEVICE_IO) must provide the appropriate character code conversions and format control characters to implement the host-independent line printer interface.

The equipment of the Perkin-Elmer 8/32 computer system does not in general impose direct requirements on components of the KAPSE, because the KAPSE runs under the host operating system OS/32. OS/32 and its device drivers handle all direct access to the machine equipment.

The limitations of the addressing of the 8/32 processor (1' Mb) places an upper bound on the size of the KAPSE, as well as any user program.

The instruction set of the 8/32 processor places requirements on the efficient design of the run-time system routines (KAPSE.RTS), in particular in the CPC of unit execution support.

3.2.2 Computer Input/Output Utilization

None of the peripheral equipment of the 4341 computer system has critical I/O timing requirements, because all devices are operated by channel processors.

Some of the <<TBD>> peripheral equipment of the 8/32 computer system may place requirements on the KAPSE.SIMPCOMP.DEVICE_IO CPC because of interrupt-per-character I/O controllers. In particular, if echoing is done by

TERMINAL_IO for all terminals, then the amount of processing per character should be kept to the order of one millisecond to support thirty 300 baud input streams.

3.2.3 Computer Interface Block Diagram

See Figure 3-1.

3.2.4 Program Interfaces

3.2.4.1 KAPSE/Tool Interface Requirements

The KAPSE/Tool interface consists of the following set of packages, detailed in section 3.3:

```
(KAPSE.SIMPCOMP) :

        Package SIMPLE_OBJECTS
        Package INTERACTIVE_IO
        Package FORMATTED_IO
        Package COMPOSITE_OBJECTS

(KAPSE.ACCECAT) :

        Package WINDOW_OBJECTS
        Package CATEGORY
        Package STRING_ATTRIBUTES
        Package NUMERIC_ATTRIBUTES
        Package ACCESS_CONTROL
        Package ACCESS_SYNCHRONIZATION

(KAPSE.MULTPROG) :

        Package PROGRAM_INVOCATION
        Package INTER_PROGRAM_COMMUNICATION
        Package DEBUGGER_INTERFACE
        Package USER_CONTEXT
        Package MAIL_SYSTEM

(KAPSE.HISTARCH) :

        Package HISTORY
        Package BACKUP_RECOVERY

(KAPSE.RTS) :

        Package DIRECT_IO
        Package SEQUENTIAL_IO
        Package TEXT_IO
        Package CALENDAR
```

These services are provided to Ada programs by a combination of a Data Base Manager/KAPSE program, and interface packages linked into the user's program. Inside the KAPSE there is one "agent" task per running program, assigned to handle all communication with that program, and perform the appropriate KAPSE calls on its behalf.

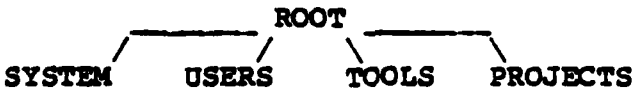
This structure is not visible to the user program, which can view the KAPSE as a set of packages linked into it. In fact, the bodies for the interface packages linked into the user programs do little more than bundle up each KAPSE/Database request into a message and then send it across to its agent task within the KAPSE. The agent task unbundles the request, and calls the appropriate package within the KAPSE, but this time, the body of the package actually does the desired work. See Figure 3-2.

The message-passing model of the KAPSE/tool interface allows the design to be hosted more easily on truly distributed processors, as well as the "virtually" distributed VM/SP system. The option also exists for several KAPSE-like programs to exist, each serving their local client programs, and communicating with each other to synchronize use of shared resources, and retrieve remotely stored data.

3.2.4.2 KAPSE User Interface

3.2.4.2.1 Overall User View of the Database

The overall structure of the database hierarchy is as follows:



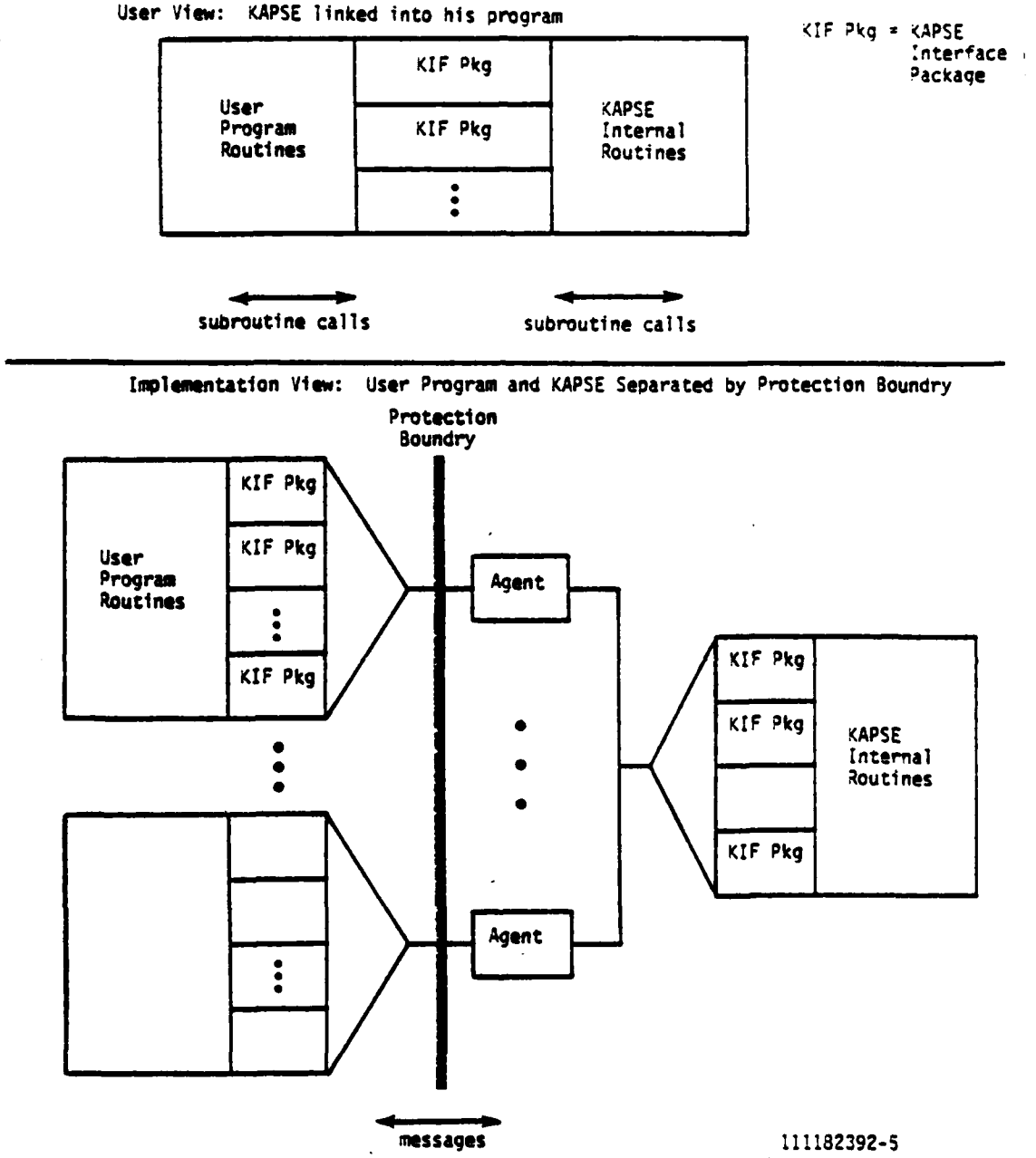
The root composite object contains four components: SYSTEM, USERS, TOOLS, and PROJECTS. All of these components are themselves composite objects. The SYSTEM composite object contains objects of interest primarily to the system manager and certain maintenance tools (eg., backup, history indices, etc.).

The USERS composite object contains the top-level composite object (directory) for each user of the MAPSE. A particular component is selected by the user's USER_NAME (see 3.3.3.6).

The TOOLS composite object contains as components all of the standard MAPSE tools (and others added by a system manager). Each component is an executable program context object, or a command language script, selected by the distinguishing attribute TOOL_NAME.

The PROJECTS composite object has the component distinguishing attribute of PROJECT, and has initial components (PROJECT=>KAPSE) and (PROJECT=>MAPSE_TOOLS) for use by MAPSE developers.

Figure 3-2,
User and Implementation View of KAPSE/Tool Interface:



3.2.4.3 Database/Tool Interface Requirements

The KAPSE database is the repository for all user data and programs, as well as the primary medium of tool to tool communication and coordination. The KAPSE database facilities

provide for the construction, organization, and partitioning of large configurations of inter-related program, data, and documentation elements. It records the nature and purpose of these elements, and allows for access control and synchronization. Finally, the KAPSE database facilities provide historical information recording the derivation and relations between the objects stored within the database, as well as sufficient information to fully reconstruct from disk or archival storage the content of old or lost source text.

3.2.4.3.1 Database Objects

The database is a collection of objects, all of which have attributes and content. These objects can be classified as follows:

a. Primitive Files — All of the data stored in the database are represented using "Primitive File" objects. Files have only pre-defined attributes like LENGTH, FIRST, ACCESS_METHOD, etc. These files are implemented using one of the built-in access methods (see 3.3.1.3), and are designed to be efficient for representing both small data items and large directories.

b. Extended Objects — The user normally works with "Extended Objects." Extended objects have a user-extendible list of labeled attributes, as well as system-defined attributes including CATEGORY, ACCESS_CONTROL, and HISTORY. The content and user-defined attributes of extended objects are either Files or Windows (see below).

c. Window Objects — "Windows" allow the user to go from one extended object (the source) to another (the target) in the database. Extended objects are available only through such windows. The window also determines the "role" a user plays in the extended object, and may limit the user to a specific "partition" of the object's content or attributes. Each extended object is available through exactly one "primary" window. Any number of "secondary" windows may also provide some (perhaps more limited) view of the object.

Each of the above objects can be further classified as follows:

a. Files fall into two classes: Simple and Composite. A Simple file is a sequence of data bytes, and provides the representation for Ada "external files," as well as for simple string- and numeric-valued attributes. A Composite file is a set of named or numbered component objects, which may themselves be either Files or Windows, and provides the representation for "directories" as well as for relations, tables, lists, sets, etc.

b. Extended objects are generally classified by the nature of their content, as either Simple Objects, Composite Objects, or Extended Window Objects. Primary windows are used to connect extended objects into enclosing composite files, thereby providing for a hierarchy of composite objects. The database as a whole is a single large composite object called the "root," whose direct components are major divisions of the database.

c. As mentioned above, windows are classified as either primary or secondary. Primary windows go from an enclosing composite object to its components. Secondary windows allow one to go from one part of the database to another part of the database (i.e. to an object not directly enclosed by the source of the window). Following only primary windows, the entire database forms a tree of extended objects. With secondary windows, the structure of the database becomes an arbitrary directed graph. See Figures 3-3 and 3-4.

The term "object" alone will generally refer to extended objects in the remainder of this specification.

3.2.4.3.2 Extended Object Attributes

The attributes of an extended object may be any kind of meta-information describing its content, purpose, version, revision, etc. As such they provide the primary means for building, organizing, and partitioning the database. Configuration management and other high-level tools will record information appropriate to their needs as attributes of objects. Lower-level tools deal primarily with the content of extended objects.

The attributes can be grouped as follows:

1. Distinguishing (name) attributes;

2. Non-distinguishing attributes:

   a. System-defined attributes (such as CATEGORY, ACCESS_CONTROL, and HISTORY);

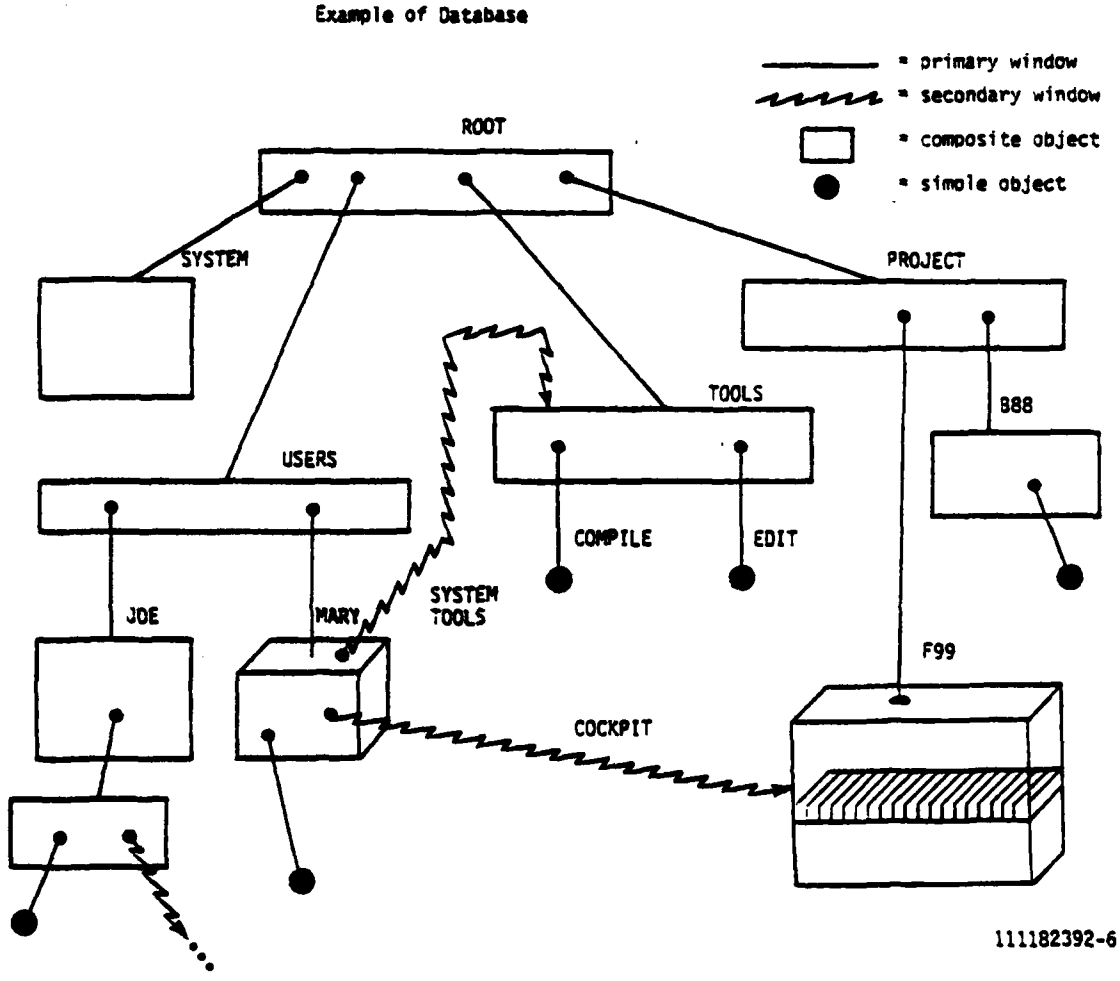
   b. Category-defined attributes;

   c. User-defined attributes;

   d. Content-defined and Path-defined attributes.

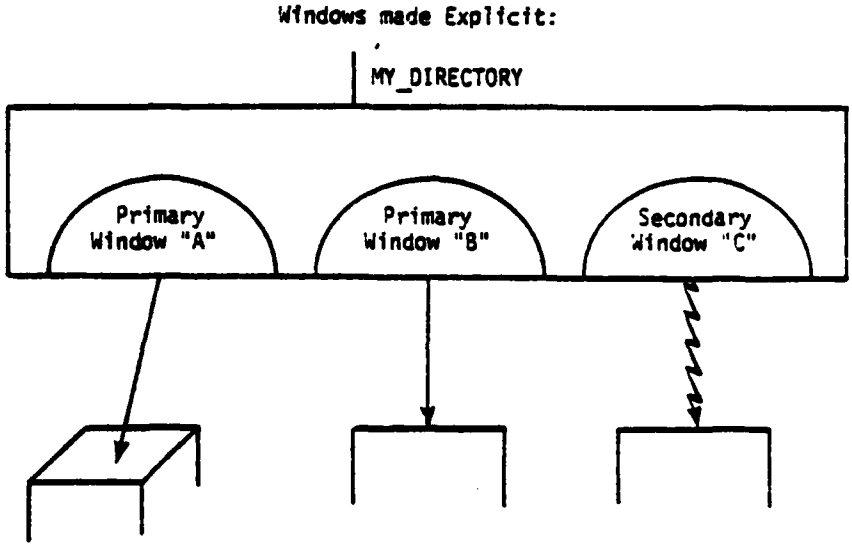
An attribute can be represented as a pair consisting of an attribute label and an attribute value. For clarity, it will be written in the unabbreviated form: label => value, to be read, label "is" value. The label of an attribute must be a string of

Figure 3-3, Example of Database:



characters which satisfy the syntax of an Ada identifier (i.e., start with a letter, and continue with letters, numbers, or underscores). The attribute value will also frequently be a string, but it may in general be any kind of File or Window.

Figure 3-4, Windows made Explicit:



Where each window may include a Role Translation Table such as:

Outside Role	Inside Role	Role Modifiers
"TESTER" "CREATOR" "WORLD" "USER"	"READER" "CREATOR" "READER" "USER"	_OWNER _READ_ONLY

111182392-2

The KAPSE supports a convenient parenthesized list notation to specify attribute label/value pairs, as shown below:

```
(PROJECT=>SHUTTLE,FUNCTIONAL_AREA=>NAVIGATION)
```

This would specify that the attribute labeled "PROJECT" has the value "SHUTTLE" and that the attribute labeled "FUNCTIONAL_AREA" has the value "NAVIGATION."

3.2.4.3.3 Naming and Partitioning with Distinguishing Attributes

Distinguishing attributes identify an object uniquely within the content of its enclosing composite object. If the component is a window, the name implicitly applies to the extended object viewed through the window.

When a composite object is created, part of its definition specifies the labels of the distinguishing attributes by which its components will be named (i.e. distinguished from one another). When a component is created within the content of this composite object, values for these distinguishing attributes must be supplied. These may be used later to select this component from the composite object. The new component may not be created if an existing component has the same list of distinguishing attribute values.

For example:

```
CREATE_COMPOSITE("COMP_OBJ",
    COMPONENT_DA=>"PROJECT FUNCTIONAL_AREA MODULE");
```

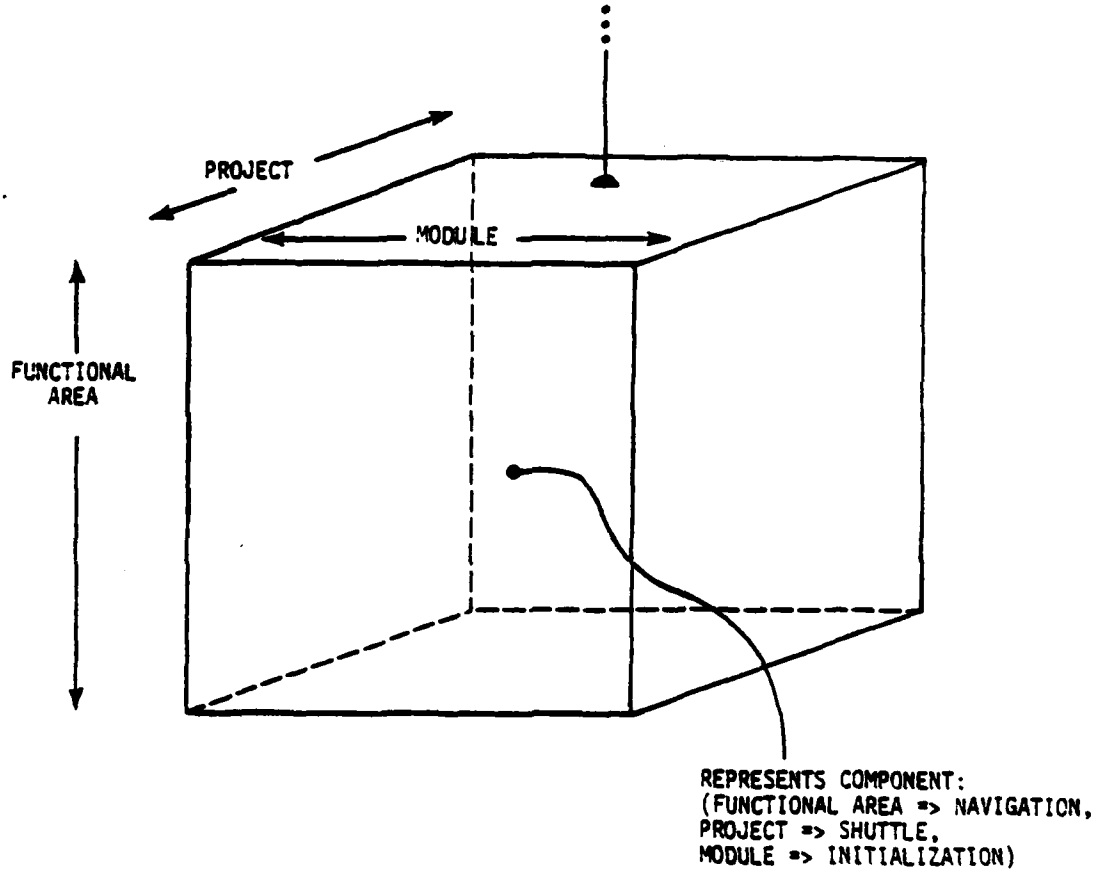
creates a composite object with the three distinguishing attributes: PROJECT, FUNCTIONAL_AREA, and MODULE. See Figure 3-5.

Now components could be created within this new composite object "named" as follows:

```
1.  (PROJECT=>SHUTTLE,FUNCTIONAL_AREA=>NAVIGATION,
        MODULE=>INITIALIZATION)

2.  (PROJECT=>SHUTTLE,FUNCTIONAL_AREA=>CONTROL,
        MODULE=>INITIALIZATION)

3.  (MODULE=>INITIALIZATION,PROJECT=>VOYAGER,
        FUNCTIONAL_AREA=>NAVIGATION)

4.  (MODULE=>INTERPOLATION,FUNCTIONAL_AREA=>NAVIGATION,
        PROJECT=>VOYAGER)
```

Two components need differ in only one of the distinguishing attribute values to be considered distinctly named (e.g., (1) and (2) above).

Figure 3-5,
Composite Object with Three Distinguishing Attributes:



Positional notation may be used instead of labeled notation, based on the ordering specified when the composite object was created:

1. SHUTTLE.NAVIGATION.INITIALIZATION

2. SHUTTLE.CONTROL.INITIALIZATION

3. VOYAGER.NAVIGATION.INITIALIZATION

4. VOYAGER.NAVIGATION.INTERPOLATION

3.2.4.3.3.1 Partitions of Composite Objects

Composite objects may be "partitioned" so as to identify a subset of the components, by specifying values for some of the distinguishing attributes of the components, while leaving others unspecified, as follows:

(PROJECT=>SHUTTLE) would include (1) and (2) from above.

(FUNCTIONAL_AREA=>NAVIGATION,MODULE=>INITIALIZATION) would include (1) and (3).

Positional notation may also be used to specify partitions, but the special value "*" must be supplied as a place holder:

.CONTROL. would include only (2)

VOYAGER.*.* would include (3) and (4)

Attributes other than distinguishing attributes may be used to specify partitions of a composite object. Non-distinguishing attributes are not ordered, and a special labeled notation is always required, using a distinguishing attribute label to "qualify" the non-distinguishing attribute label. Here is an example of a single partition specification giving values for both kinds of attributes:

(FUNCTIONAL_AREA=>NAVIGATION,MODULE'PRIORITY=>HIGH)

This partition would include (1), (3), and (4) from above only if their current value for a non-distinguishing attribute labeled "PRIORITY" were "HIGH." If a non-distinguishing user-defined attribute has never been specified for an object, the value is taken to be the null string.

3.2.4.3.3.2 Object Pathnames

An object which is a sub-sub-component of a composite object, can be identified relative to the upper level composite object by specifying a sequence of two component names separated by periods (each component name may itself be a sequence of distinguishing attributes). This process may be continued leading to the general concept of pathname, where a subcomponent of an object is identified by a path from the composite through the intermediary composites, leading finally to the desired object. Each object in the entire database can be uniquely identified by the path to it which follows only primary windows from the root composite object. For example:

```
USERS.JONES.(SUBPROJECT=>VOYAGER,PHASE=>DOCUMENTATION).A.B
```

could be the pathname from the root to some sub-sub object. Notice that the parenthesized, labeled notation and the positional notation may be mixed if joined by dots, with the parenthesized portions interpreted relative to the point reached by the preceding part of the pathname.

Pathnames may also be used to identify attributes rather than components of the content. In these cases, the apostrophe (single quote, "tic") is used to distinguish an attribute label from a component name. The attribute named may itself be a complex object, in which case the pathname may continue after the attribute label with further component or attribute names. For example:

```
SYSTEM.PRINT'QUEUE'FIRST.BODY
```

might be the pathname to an object which is the body of a listing that is first on the system print queue.

3.2.4.3.3.3 Secondary Windows and Pathnames

Object pathnames may be specified which do not follow strictly down the hierarchy of composite objects and attributes, by traversing secondary windows. When a secondary window is encountered in a pathname, the rest of the pathname is interpreted relative to the partition of the extended object referred to by the window.

When a secondary window is created, the name of the target object and the partition limitation, if any, are specified. In addition, the user may specify further limits on the allowed range of access to operations on the attributes and content of the extended object. Secondary windows are the means by which a user may delegate access to and/or responsibility for parts of the database to other users.

The access limitations of a window are expressed as an abstract role to which users of the window are limited. The role is identified by an ASCII STRING, like "MANAGER", "READER", or "DEVELOPER," which must be one of the roles listed in the ROLES attribute of the viewed extended object. Within the partition of the extended object designated by the window, the user is limited to the operations allowed to the role by the object's ACCESS_CONTROL attribute (see below). Multiple windows may exist specifying different roles relative to the same partition.

For example, consider a pathname (starting at the root), "USERS.JOHN.MARYS_LIB," that identifies a secondary window whose target has the full name "USERS.MARY.LIB." Then the pathname, "USERS.JOHN.MARYS_LIB.Q.R," would identify the same object that the pathname, "USERS.MARY.LIB.Q.R," identified. However, because different paths were followed to the object, different roles, and hence different access rights, may be held at the object.

3.2.4.3.4 Non-distinguishing Attributes

3.2.4.3.4.1 System-Defined Attributes

Several attribute labels are predefined by the KAPSE to have special meanings, and hence are not available as labels for user-defined attributes. They include among others:

a. CATEGORY_DESCRIPTOR

b. CATEGORY

c. USER_DEFINED_ATTRIBUTES

d. CONTENT

e. ACCESS_CONTROL

f. HISTORY

Such system-defined attribute labels will be capitalized in discussions that follow.

3.2.4.3.4.2 Category Descriptor and Category-Defined Attributes

Every extended object has a CATEGORY and a CATEGORY_DESCRIPTOR. The CATEGORY_DESCRIPTOR attribute is like a type descriptor for a database object, and specifies the class and structure of the object, and may place constraints on the values of particular attributes. The CATEGORY attribute is a string, like "ADA_SOURCE" or "TEST_SCRIPT," acting as the identifier for the type of the database object. The CATEGORY attribute is purposely similar to the notion of "type" in high

level languages and provides the user with much the same kind of data abstraction and structuring.

The category descriptor includes a table of attribute descriptors, keyed by the category-defined attribute labels. Each attribute descriptor specifies whether the attribute is a constant across the category, or defined as a particular element of each object of the category (a "variable"). For category-defined attributes that are variable, other limitations may be placed on the range of permissible values, as well as a default value when the value has not been specified.

The category descriptor of an object is set when the object is created, and may not be changed. Nevertheless, the effect of changing the descriptor can be effected by creating a new object with the desired changed descriptor, and then copying the content and appropriate attributes into the new object. During this process, the KAPSE can verify that the values of the attributes do not violate any constraints implied by the new descriptor.

3.2.4.3.4.3 User-defined Attributes

The CATEGORY_DESCRIPTOR defines all of the category-defined attributes for the object. Nevertheless, the KAPSE supports a convention whereby the single system-defined attribute labeled "USER_DEFINED_ATTRIBUTES" may be used to store values of attributes not explicitly defined in the category. Hence, the actual list of legal attribute labels is effectively unbounded for extended objects.

User-defined attributes are created when assigned a non-null value (which may be any file or window), and are deleted when assigned a null value. Individual attributes are retrieved using the attribute label as the key into the composite USER_DEFINED_ATTRIBUTES file. For instance, OBJECT'PURPOSE is equivalent to OBJECT'USER_DEFINED_ATTRIBUTES.PURPOSE if PURPOSE is a user-defined attribute label.

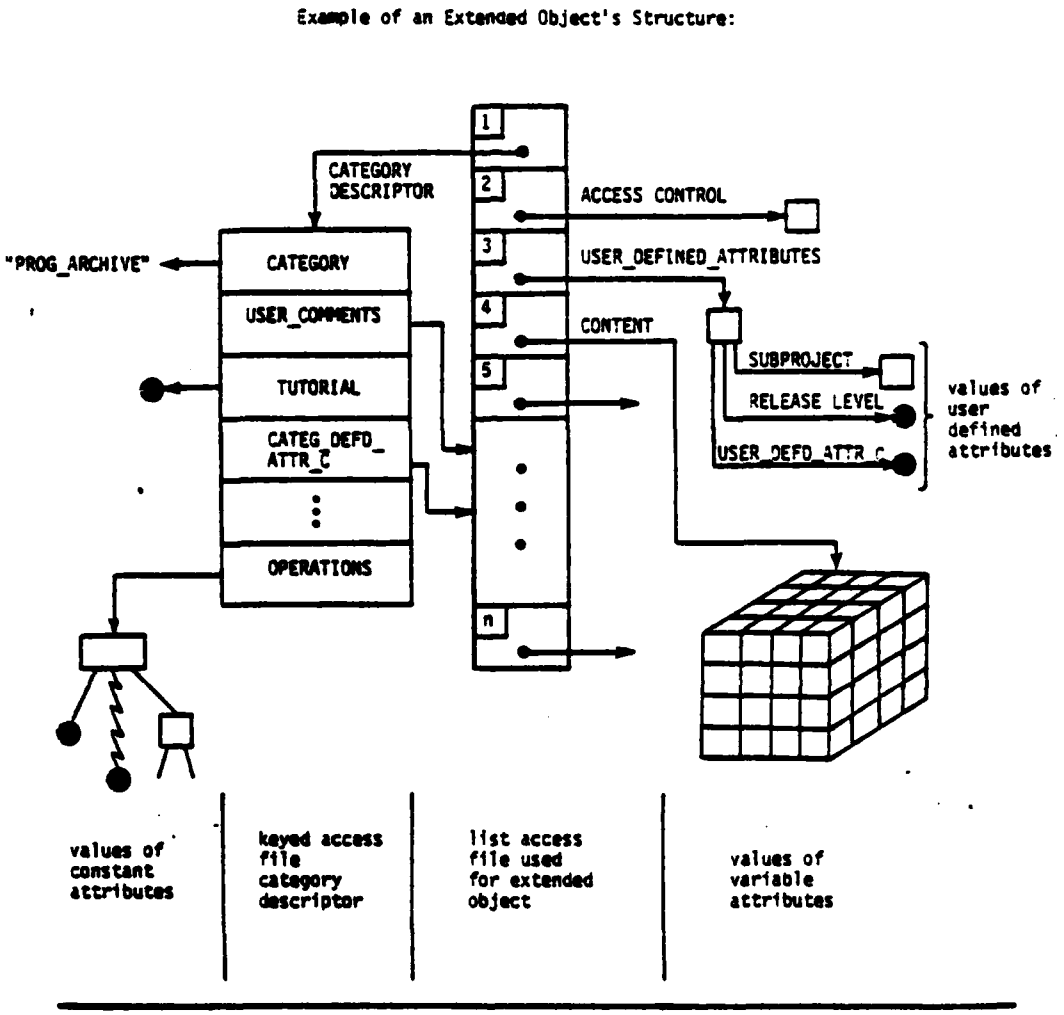
See Figure 3-6 for an example of an extended object with category- and user-defined attributes.

3.2.4.3.4.4 Other Attributes

3.2.4.3.4.4.1 Content-defined Attributes

For convenience, the attributes of the content of an extended object are accessible as though they were attributes of the extended object itself. Content-defined attributes are hidden by other pre-existing attributes of the same label. The attribute label may be prefixed by "CONTENT'" to override this. For example, OBJECT'LENGTH is equivalent to OBJECT'CONTENT'LENGTH if LENGTH is a content-defined attribute label.

Figure 3-6, Example of an Extended Object's Structure:



Note that the labels for the attributes in element positions 1,2,3,and 4 are system defined.

111192392-10

To hide existing content-defined attributes with a new user-defined attribute of the same label, the attribute USER_DEFINED_ATTRIBUTES must be referenced explicitly the first time.

3.2.4.3.4.4.2 Target-defined Attributes

Again for convenience, the attributes of the target of a window are accessible as though they were attributes of the window itself. Windows have no nameable attributes themselves. Certain special primitives (see 3.3.2.1) and path-defined attributes (see below) are available to query and adjust information about a window.

3.2.4.3.4.4.3 Path-defined Attributes

The path-defined attributes CURRENT_ROLES, CURRENT_MODIFIERS, CURRENT_RIGHTS, CURRENT_OPERATIONS, CURRENT_CHANNELS, and CURRENT_PARTITION, are accessible to reveal the access available at a particular moment via a particular path. These attributes are not stored on any object, but are rather derived from the path followed to reach the object.

3.2.4.3.5 Uses for Composite Objects

Composite objects are used for the normal concept of a user directory. However, because of the ability to use multiple distinguishing attributes, to define partitions, and to operate and control the objects as a whole with the various KAPSE database primitives, composite objects can serve many other purposes as well.

3.2.4.3.5.1 Configurations as Composite Objects

Composite objects can be used to hold the set of component objects which represent a particular configuration of a system. The configuration can also be adjusted as necessary by Ada programs. New components can be created, existing components can be modified or deleted. The configuration, because it is a single composite object, can be copied as a unit. The structure of the configuration can be laid out and controlled by a category descriptor, and access control can be applied to the configuration as a whole, or to its individual parts.

3.2.4.3.5.2 Program Context Object

Each program running in the MAPSE has associated with it a single composite object called its program context object, or simply "context object." It is through the context object that Ada programs gain access to the rest of the database. All object pathnames begin in the context object, and then go through windows to other more permanent parts of the database.

The context object is normally deleted after the program finishes execution, and its results and status have been reported

to its invoker. Components and attributes of this context object may be simple objects (temporary files), composite objects (a set of temporary objects), or windows on the more permanent parts of the database. All context objects are composite objects using a single distinguishing attribute labeled LOCAL_NAME for their components.

When an Ada program creates or opens an object in the database, it specifies the pathname. If the pathname begins with a dot or a tic, then the rest of the path is interpreted relative to the context object.

If the pathname does not begin with a dot or a tic, the KAPSE uses the window attribute labeled CURRENT_DATA of the context object, and interprets the path relative to that window. In effect, it is as though "'CURRENT_DATA." were inserted at the front of the pathname.

When a program is to be invoked from some existing running program (i.e. the compiler being initiated from the command language processor), a new context object is created, initialized with a window attribute called "'PROGRAM" back on the executable program object, and with other window attributes and parameters inherited from the invoker and its context object. This new (sub)context is by default created as a component of the content of the invoker's context object, allowing the invoker to refer to the subcontext during its execution by its LOCAL_NAME. See Figure 3-7 and section 3.3.3.3.

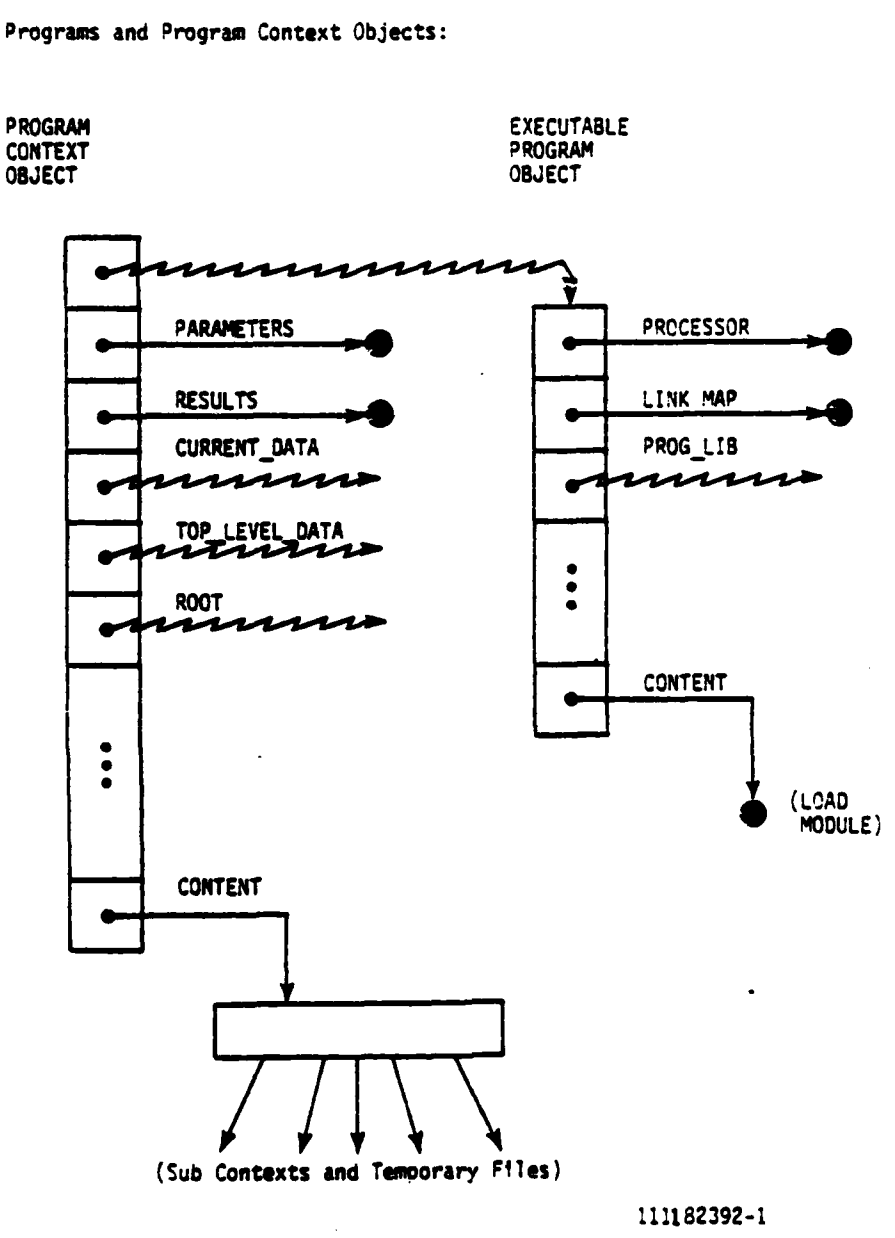
3.2.4.3.5.3 Private Objects

The KAPSE supports the creation and controlled manipulation of private objects, encapsulated abstract data objects analogous to Ada objects of private type. Private objects are managed by one or more "trusted" programs, instead of being directly accessible to a user through the normal database I/O operations. For example, the KAPSE mail system allows users to send and receive mail using private objects called mailboxes, without giving users the ability to corrupt the internal structure of the mailboxes.

Two implementations of private objects are supported by the KAPSE. The first takes advantage of the KAPSE's access control facilities to implement an object manipulated by programs local to the object. The second uses access control combined with the INVOKE_OPERATION primitive to implement an object manipulated by programs ("operations") external to the object.

The simpler kind of private object is a composite object with one data component, and a number of executable program components, each with privileged window attributes back on the data component. To most external windows, only "execute" access

Figure 3-7, Programs and Program Context Objects:



is given to the program components, and no direct access to the data component. In this way, direct access to the data is denied most users, but they can access the data via the programs supplied in the private object.

When executed, the context objects for the program components will have their 'PROGRAM window back on their program objects. With that, and the privileged window attributes of the program objects, they can implement, on the user's behalf, useful high-level operations on the data component.

This kind of private object requires no special handling by the KAPSE, as the 'PROGRAM window attribute is a standard feature available to all programs. This kind of encapsulation is analogous to that provided by an Ada package with externally visible procedures, but with private data.

The second kind of private object does not contain direct program components, but rather has an attribute labeled "'OPERATIONS" which is a window on a composite object where the trusted programs reside. These trusted program objects cannot be attributed with windows on the private object, because they do not know in advance which object is to be manipulated. Instead, the KAPSE provides an INVOKE_OPERATION primitive which, given the name of an operation as a simple string, and a path to the private object, will look up the operation in the composite referred to by the designated object's OPERATIONS attribute. If there, it will initiate the selected program with a privileged window in its context of the name "'IMPLICIT_OBJECT" referring to the private object.

By restricting access to the object so that the user can only invoke the "operations" (see Operate access right, below), the private object is analogous to an instance of a "private" type of an Ada package.

It should be noted that an object need not be a totally private object to have an OPERATIONS attribute; the ability to invoke an operation on an object as well as gain other direct access can be useful.

3.2.4.3.6 Access Control for Extended Objects

Access control within the KAPSE is based on the concept of "role." Each extended object may define its own set of abstract roles. A role represents a logical set of participants in the access and manipulation of an extended object. For example, a project manager might wish to define roles for a REVIEWER, a TESTER, a PROGRAMMER, and a MANAGER within a particular extended object.

The manager would then set up the ACCESS_CONTROL attribute, which is associated with every extended object, to define the set of "concrete access rights" which each role may use in the object. For example, (s)he could give read access to the REVIEWER, read and execute access to the TESTER, and read, write, and execute to the PROGRAMMER, while reserving the special _OVERSEER modifier for the MANAGER role, as shown below:

Role	Concrete Access Rights
REVIEWER	Read
TESTER	Read, Execute
PROGRAMMER	Read, Write, Execute
MANAGER	Read (plus _OVERSEER modifier -- see below)

3.2.4.3.6.1 Abstract Roles

The set of roles for an extended object is defined by the ROLES attribute, an indexed list which serves to provide a mapping between role indices and role names. The role indices are stored internally (as bit positions, generally) instead of role names in all windows and attributes which refer to roles. Role names are provided externally for the user's benefit. Certain role indices are reserved for system-defined roles (e.g. SYSTEM, CALLER, SUBCONTEXT).

3.2.4.3.6.2 Access Control Attribute

The ACCESS_CONTROL attribute for an extended object is a list of access control elements. An element specifies the following for a specific role:

a. A set of available concrete rights;

b. A set of available operations;

c. A set of available communication channels;

These rights are possibly limited or extended by partition limitations and role modifiers (see below). Each of the above sets are represented with bit vectors, using the appropriate index (role index, right number, operation index, channel index).

3.2.4.3.6.3 Concrete Rights

The following concrete access rights are pre-defined for the AIE:

a. Read — This right is required to be able to read the files which make up the content and attributes of the extended object. It also controls whether one can go through any of the enclosed windows.

b. Add — This right is required to be able to add information to any file or window, including appending to a simple file and creating new elements of a composite file.

c. Delete — This right is required to be able to delete information from any file or window, including removing from the head or tail of a simple file (aka "consuming"), and

deleting existing elements of a composite file. Both add and delete rights are required for random-access writing of a simple file, and replacing of existing elements of a composite file.

d. Execute This right is required to be allowed to execute the content of the extended object as a program. A PROCESSOR attribute must also be defined for the extended object. The PROCESSOR attribute specifies the processor which will interpret the content of the extended object. The attribute value is the name of the target machine for compiled code, or a window on the interpreter for a command language script.

e. Operate This right is required to be allowed to invoke any of the operations of the extended object. An OPERATIONS attribute must also be defined for the extended object, and the operation index must be included in the available-operation set.

f. Communicate This right, which is only relevant to running program context objects, is required for any communication with or control over the running program. To communicate on a program-defined channel, a CHANNELS attribute (which associates a channel index with each of the channel names accepted by the program) must also be defined for the context object. The system-defined channels (e.g. "_CONTROL," "_DEBUG") are assigned pre-defined indices. The system- or program-defined channel index must be included in the available channel set.

Operate and communicate are mutually exclusive -- communicate only applies to running program contexts, operate only applies to data objects with an OPERATIONS attribute. The operation index set, and the channel index set may therefore be defined by the same set of bits, interpreted appropriately.

3.2.4.3.6.4 Windows and Access Control

Windows specify the roles (and role modifiers, see below) to be used within an extended object, in terms of the roles used in the extended object enclosing the window. The concrete right "read" is also needed to "go through" the window at all.

The role "translation" is expressed as follows:

a. A set of outside roles which retain the same roles inside the extended object (they "translate" into themselves).

b. A list of translations. The first part of each translation identifies a set of outside roles; the second part identifies the set of internal roles and modifiers inherited by any member of the first set. This allows for the

possibility that a user (typically a manager) may have several roles at the same time. The set of internal roles and modifiers resulting from a window translation is the union of those provided to each of the outside roles held on entering the window.

3.2.4.3.6.5 Role Modifiers

A set of role "modifiers" are predefined for all extended objects. These modifiers determine extra rights and limitations associated with the roles held:

a. _OWNER This modifier means that the ACCESS_CONTROL attribute of the extended object may be edited (see 3.3.2.3) This modifier is always translated into _OVERSEER on going through a primary window, and is lost on going through a secondary window. This modifier is automatically given to the creator of an object.

b. _OVERSEER This modifier means that the ACCESS_CONTROL may be edited if one of the roles held already has read access as defined by the existing ACCESS_CONTROL attribute. This modifier is preserved on going through a primary window, but is lost on going through a secondary window. This modifier is automatically given to the copyer of an object, on the copy only.

c. _READ_ONLY This modifier means that no modifications may be made to the object or any of its components, independent of any rights granted by access control attributes. This modifier is preserved on going through a window. This modifier is automatically given when the partition of the window includes non-distinguishing attribute limitations.

3.2.4.3.7 Primary Windows and Extended Objects

When an extended object is to be created, a primary window is first created with the designated name, and then the new object is created as its target. The creator must have the "add" concrete right on the file in which this primary window is being implicitly created. If an existing object is being replaced, the creator must have the "delete" concrete right as well. The ROLES and ACCESS_CONTROL attributes of the extended object are initialized by default to be the same as the enclosing extended object, unless overridden by additional parameters to the CREATE primitive. The implicitly created primary window is initialized to allow external roles to keep their role internally, but also to give the _OWNER modifier to the set of roles held by the creator.

When an extended object is copied, a new primary window is created to control access to the copy. The copy is otherwise

identical to the original. The translation table of the new primary window is set up to translate the copyer's roles outside this new window, to the same roles already held by the copyer inside the original extended object. This allows a large extended object to be safely copied, without first having to verify that the copyer has read access to all of its sub-components. The copyer's roles are given the _OVERSEER modifier for use in the new copy, so that its ACCESS_CONTROL attribute can be adjusted.

The _OWNER and _OVERSEER modifiers are important because of the special nature of the ACCESS_CONTROL attribute. It is the attribute which defines the access rights to the object as a whole, and hence, any role which can edit the access control attribute has effectively unlimited access to the object.

The _OWNER modifier is given to the creator of an extended object. The creator may then adjust the ACCESS_CONTROL arbitrarily. On the other hand, when an existing extended object is copied, only the _OVERSEER modifier is given to the copyer on the new copy. What this accomplishes is that the copyer can adjust the access control of the copy only in components of it that he or she could already read.

3.2.4.3.8 Secondary Windows and Extended Objects

As mentioned above, secondary windows may be created on an extended object. Secondary windows have in addition to their role translation table:

a. An object designator of the target;

b. An optional partition limitation.

3.2.4.3.8.1 Role Translation

When the secondary window is created, the translation table is set up, by default, to translate the roles held outside the new window, to the roles (and role modifiers) already held by the creator inside the extended object. If desired, the creator may further limit the roles and modifiers inherited via the new window.

3.2.4.3.8.2 Target Object Designator

The target object designator of a secondary window consists of a common ancestor label and a window key (see below). The common ancestor label uniquely identifies an extended object which must enclose both the window and its target. The common ancestor's label, stored in the system-defined attribute NODE_LABEL, is assigned automatically when the first window wanting to use it is created. These NODE_LABELs are large

integers to minimize collisions (it should be noted, however, that even given collisions the KAPSE will find a suitable, if not optimal, common ancestor). The NODE_LABEL of the root of the database is distinguished (e.g. always 1) for efficiency. The root can always be used as the common ancestor as a last resort.

The window key is assigned when the window is created, to be unique among all windows using the same common ancestor, by advancing the value of the attribute LAST_WINDOW_KEY of the common ancestor. These keys are never reused. Periodically, the KAPSE will do a "garbage collection" to identify keys that have no further references and compress the key space.

The window key is used to select an element of the WINDOW_XREF attribute of the common ancestor. The element selected is a file with information relevant to the window, as follows:

a. ORIGINAL_WINDOW — The path from the common ancestor back down to the original window created. This attribute does not change, hence it in effect "names" the window for all time in terms that are meaningful to the creator of the window. If the creator decides later to revoke the window, then he can do so by specifying the value of this attribute in the revoke operation.

b. TARGET — The path down to the target. It is an error if this path does not "end" on an extended object, or passes through secondary windows.

c. PARTITION — The partition of the target visible to users of the window. By default, this partition is all of the target.

d. ROLE_SET — The set of all roles available to users of the window, as based on the original translation table. This set is represented as a bit vector.

e. MODIFIER_SET — The set of modifiers for the users of the window, as specified in the original translation table. This set is represented as a bit vector.

f. PARENTS — A list identifying all the parent windows of this window (see 3.2.4.3.8.4 below).

g. TRANSITORY — A flag, which indicates that this window will soon disappear and may not be listed as the parent of any window derived from it (see 3.2.4.3.8.4 below).

h. HAS_CHILDREN — A flag, indicating that a window has been created with this as one of its parents.

i. REVOKED — A flag, indicating that this window, and all of its copies, have been revoked, but their children have not. The children are then treated as direct children of the parent windows.

It is important to understand that window creation is distinguished from window copying. No additional WINDOW_XREF elements are created for copies of windows (in particular those copies due to the copying of a large enclosing extended object). On the other hand, each creation of a new window involves the creation of an element of the chosen common ancestor's WINDOW_XREF attribute.

3.2.4.3.8.3 The Common Ancestor

When a window is created, the target object is designated relative to some "common ancestor" extended object. This allows a large Extended object enclosing a window, its target, and the common ancestor to be copied and preserve the window/target relationship in the new extended object. Alternatively, if the copied object does not include the common ancestor, then the new copy of the window will continue to refer to the original target. This flexibility means that the judicious choice of a common ancestor can determine whether a window is considered to point to a specific "absolute" object, or just to an object in some local relationship with it.

Besides enclosing the window and its target, the common ancestor must enclose the common ancestors of all of the parent windows (see below). If the common ancestor is not explicitly specified, the nearest ancestor satisfying these requirements is used.

3.2.4.3.8.4 The Parents of a Window

Whenever a secondary window is created, it may be recording roles which were obtained as the result of traversing some pre-existing secondary windows. The creator must specify a path to the target (TARGET_PATH), and a path to where the window should be created (WINDOW_PATH). Both of these paths as usual start in the context object of the creating program. The new window is defined to be "derived" from all those windows traversed by the TARGET_PATH which were not also traversed by the WINDOW_PATH. This definition is based on the theory that if the TARGET_PATH and the WINDOW_PATH start out the same, those windows traversed along this common part will still have to be traversed to reach the starting point of the newly created window (i.e. WINDOW_PATH).

The "parents" of the new window are defined to be the union of:

a. Those windows from which it was "derived" which don't have the TRANSITORY flag set;

b. The parents of those windows which do have the TRANSITORY flag set;

The "parent" relationship forms a directed acyclic graph recording window creation dependence. This forms the basis for window revocation rights (see below).

The path to the common ancestor of each of these parents, along with the parent's key at the common ancestor, are recorded in the PARENTS component of the WINDOW_XREF element for the newly created window.

3.2.4.3.8.5 Transitory Windows

The TRANSITORY flag is used to prevent permanent dependence on temporary windows created simply for focusing on a part of the data base. It is envisioned that an interactive user will move through the data base by "changing view" from one location to another traversing both primary and secondary windows. A transitory window (i.e. 'CURRENT_DATA) will record each new "view". The windows derived from the transitory window will take as parents not the transitory window itself, but rather the parents of the transitory window, allowing the transitory window to be deleted without affecting windows derived from it. Interactive users may walk around the database until they have precisely the view desired and then derive a more permanent window from the transitory window.

3.2.4.3.8.6 Revoking a secondary window

After creating a window, there may come a time when the rights thereby granted are to be revoked. The right to revoke a window is limited to those with _OVERSEER or _OWNER role modifiers at one of the following "places":

a. At the location of the window itself.

b. At the location(s) of the parent window(s).

c. At the target extended object, but only if none of the parent windows end at the same target (this represents a definition of the direct "children" of the target).

A window may be revoked in only two ways:

a. The window and all of its descendants may be revoked;
   or

b. The window may be revoked, and its children remain to become adopted children of its parents.

Note that it is only possible to revoke a direct child window. It is not possible to selectively revoke "grand-children" windows, without first revoking the responsible parent. This provides sufficient flexibility, without sacrificing reasonable accountability rules (see [Warshall80] for a further discussion of these issues).

3.2.4.3.9 History Recording and Management

From the point of view of history, two significantly different kinds of extended objects exist in the database: source objects and derived objects. Source objects are those with content produced, in general, by a human using a text editor. Derived objects, text or otherwise, are those produced as the output of other tools or user programs, with little or no direct input from the user other than parameters.

The history attribute is designed to uniquely identify a particular "state" of an extended object's content, and to record enough raw data to support present and future monitoring, analysis, and rederivation tools.

In the case of a source object, the history attribute refers to a "source archive" wherein an efficient representation of multiple states (revisions or versions) of the same basic text may be stored. The history attribute consists of a window on the source archive, and an index used to locate the pieces of text that make up this particular state of the object. Given a copy of the history attribute of a source object, it is possible to reconstruct the original text (subject to access control).

When an object is first created, it is considered by default to be a derived object. It may then be explicitly identified as a source object, at which time, it may be added to an existing source archive, or used to create a new source archive. When added to an existing source archive, the source object is assigned the next sequential state index. With a new archive, the source object is assigned state number one.

The history attribute of a "derived" object consists of a window on a program invocation "script," and an index indicating which output of the program gave this state of the object. Every time a program is invoked, a script is created to record its parameters, when it was invoked, an list of windows on the objects manipulated by the program, and copies of their history attributes (including the TERMINAL_INPUT object -- see 3.3.3.5). If the program modifies no objects, the script may be deleted after the program completes. Otherwise, the script must be saved permanently, and the history attributes of each of the modified objects must be updated to point to the script.

The window mechanism keeps track of references to history scripts and archives. Periodically, the contents of source

archives and scripts that have not been referenced recently may be dumped to tape through a KAPSE service (see 3.3.4.1). Nevertheless, a "stub" remains to keep track of where the history has been saved off-line. A user may explicitly request re-activation of specific scripts or archives. Even recently referenced archives or scripts may be written to tape to ensure that the tape contains an internally self-consistent representation of history. However, these history elements are left on-line as well.

In addition to the data mentioned above, each history script and source archive records the date and time, as well as the USER NAME, when the program execution or source archiving occurred (see 3.3.4.1).

3.2.4.4 KAPSE/Host Interface -- VM/SP and OS/32

The KAPSE is designed to be as host-independent as possible. This is accomplished by defining KAPSE/Host interface packages which provide for the rest of the KAPSE a uniform interface to the host. Only the bodies of these interface packages will need re-writing when re-hosting. Also, any embedded machine code is restricted to the KAPSE/Host interface packages.

The KAPSE/Host interface packages, detailed in section 3.3, are as follows:

```
(KAPSE.SIMPCOMP) :

        Package PHYS_BLOCK_IO
        Package DEVICE_IO

(KAPSE.MULTPROG) :

        Package PROGRAM_LOADING
        Package KAPSE_PROGRAM_COMMUNICATION
        Package KAPSE_KAPSE_COMMUNICATION

(KAPSE.RTS) :
        (Most of it, except the
          "Language-Defined Packages" CPC)
```

Note that, in a sense, these packages represent the opposite "side" of the KAPSE from the KAPSE/Tool interface packages identified in section 3.2.4.1.

3.2.4.4.1 Overall Architecture

The overall architecture provided by the KAPSE/Host interface packages, using whatever host facilities are appropriate, is a number of independently executing Ada programs

running concurrently on the host machine. Each independent Ada program has its own run-time system, including an Ada task scheduler.

The KAPSE/Host interface packages implement (with help from the host) device drivers, as well as the loading, timesharing, memory management, and swapping of the independent programs. The KAPSE/Host interface packages also provide a low-level communication path between the KAPSE and each user program (package KAPSE_PROGRAM_COMMUNICATION), and between two KAPSEs on separate (virtual) machines (package KAPSE_KAPSE_COMMUNICATION).

The KAPSE itself is an Ada program, using a specialized version of the Ada run-time system lacking the high-level IO packages, and supporting the connection of entries of tasks within the KAPSE/Host interface packages to real hardware interrupts (see 3.3.5.3).

The communication path from the user Ada program to the KAPSE is analogous to a "system call" or SVC. Except for this communication path, the KAPSE/Host interface packages entirely isolate the user Ada programs from one another, from the KAPSE, and from the host (using hardware protection where possible).

In summary, the KAPSE/Host interface packages insulate the KAPSE from the idiosyncrasies of the host system facilities. The rest of the KAPSE, in turn, implements the high-level KAPSE/Tool interfaces in terms of these low-level host-independent interfaces. In addition, the KAPSE/Host interface packages prevent the application programs running under the KAPSE from accessing the host facilities directly, thus ensuring that the KAPSE Database is not contaminated.

3.2.4.4.1.1 IBM VM/SP

This overall logical architecture is implemented on top of the VM/SP system using multiple virtual machines [IBM81], each with its own KAPSE program. A particular user is allowed to DIAL into one of the running virtual machines, or to IPL his own (or his project's) if it is not already running. After IPL, the KAPSE begins running and spawns multiple LOGIN programs within its virtual machine to handle the terminal associated with the IPL, and each terminal which connects later via DIAL.

After connecting via IPL or DIAL, the user must LOGIN by providing a user name and password. If accepted, LOGIN then invokes the command processor identified in the user's INITIAL_PROGRAM_CONTEXT attribute of his top-level directory. The additional programs initiated by the command processor will share the same virtual machine with the KAPSE and those of other simultaneous users of the virtual machine. The multiple programs within a single virtual machine are managed by the KAPSE/Host

interface package PROGRAM_LOADING.

To provide access to each other's databases, multiple KAPSEs on the same physical machine may communicate via the KAPSE/Host interface package KAPSE_KAPSE_COMMUNICATION. This communication between virtual machines is implemented using the Virtual Machine Communication Facility (VMCF) or the Inter-User Communication Vehicle (IUCV), both of which are high-band-width memory-to-memory data paths provided by the VM/SP Control Program.

3.2.4.4.1.2 Perkin-Elmer OS/32

The same overall logical architecture is implemented differently on top of the OS/32 system, by placing each independently executing Ada program in its own OS/32 task. User programs execute in a mode whereby the only OS/32 SVC 6 system calls they can perform are inter-task communication. They are not permitted to directly stop, start, or otherwise interfere with other tasks (NOCON mode) [PE79].

The KAPSE runs in its own OS/32 task without NOCON mode, allowing the KAPSE/Host interface packages access to all OS/32 system calls. They initiate all physical I/O, including terminal and disk, and thereby can optimize physical disk access and provide a central buffer cache. All OS/32 tasks communicate using the standard OS/32 inter-task communication primitives, a memory-to-memory queue-based data path [PE79].

3.2.4.5 Compiler/Run-time System Interface

The Ada compiler [AIE(1).COMP(1)], like any tool written in Ada, depends on its own run-time system (KAPSE.RTS) implicitly for proper execution. In addition, the middle and back end of the compiler (COMP.MID, COMP.BEND) depend on the interface presented in general to compiled code by the run-time support routines, because they expand and generate the code for Ada language constructs which implicitly use those interfaces.

Each of the CPC's within the KAPSE.RTS CPCI provide a set of routines, some expanded in-line, others called out-of-line within compiled code. The interfaces to these routines, as known to the compiler, are detailed in each of the discussions in section 3.3.5.

3.2.4.6 Linker/Loader interface

The linker [AIE(1).PIF(1)] depends on the load module format, defined by the loader within the package PROGRAM_LOADING of the KAPSE/Host interface, CPCI KAPSE.MULTPROG. This interface remains to be fully specified, and may vary from host to host.

In general, a load module will be a single direct-access file, written using the language-defined package DIRECT_IO (KAPSE.RTS). The information within the load module is sufficient to identify the size, layout, and initialization of virtual memory for the program's code and static data, as well as identify where execution is to begin, and what initial stack allocations are appropriate.

3.2.5 Function Description

The KAPSE consists of five Computer Program Configuration Items (CPCI's):

a. KAPSE.SIMPCOMP

b. KAPSE.ACCECAT

c. KAPSE.MULTPROG

d. KAPSE.HISTARCH

e. KAPSE.RTS

A brief discussion of each of these CPCI's follows. See Figure 3-8 for an overall CPCI dependency diagram.

3.2.5.1 Simple and Composite Objects (KAPSE.SIMPCOMP)

This CPCI defines simple and composite objects; it defines the techniques used to implement objects as well as the operations that can be performed on objects. Access methods and data clumps define the techniques used to implement simple objects. Simple objects correspond to files on a typical operating system. Composite objects are collections of simple objects. A traditional directory is an example of a composite object. Also defined are the routines that do physical input and output (I/O). Block I/O defines read and write routines between a program and the disk; device I/O defines read and write routines between a program and an interactive terminal.

A short discussion of each of the CPC's that comprise this CPCI follows. See Figure 3-9 for the inter-CPC dependencies within this CPCI.

3.2.5.1.1 BLOCK IO

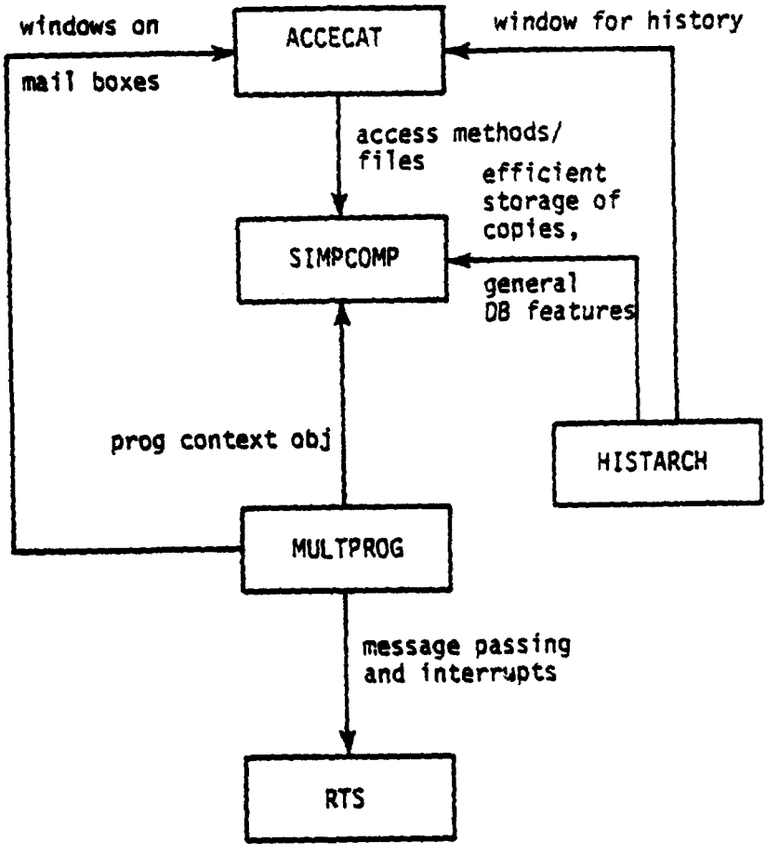
BLOCK IO defines routines to read from and write to the disk in fixed size blocks.

3.2.5.1.2 DEVICE IO

DEVICE IO defines routines to read from and write to an interactive terminal.

Figure 3-8, Top Level CPCI Implementation Dependencies:

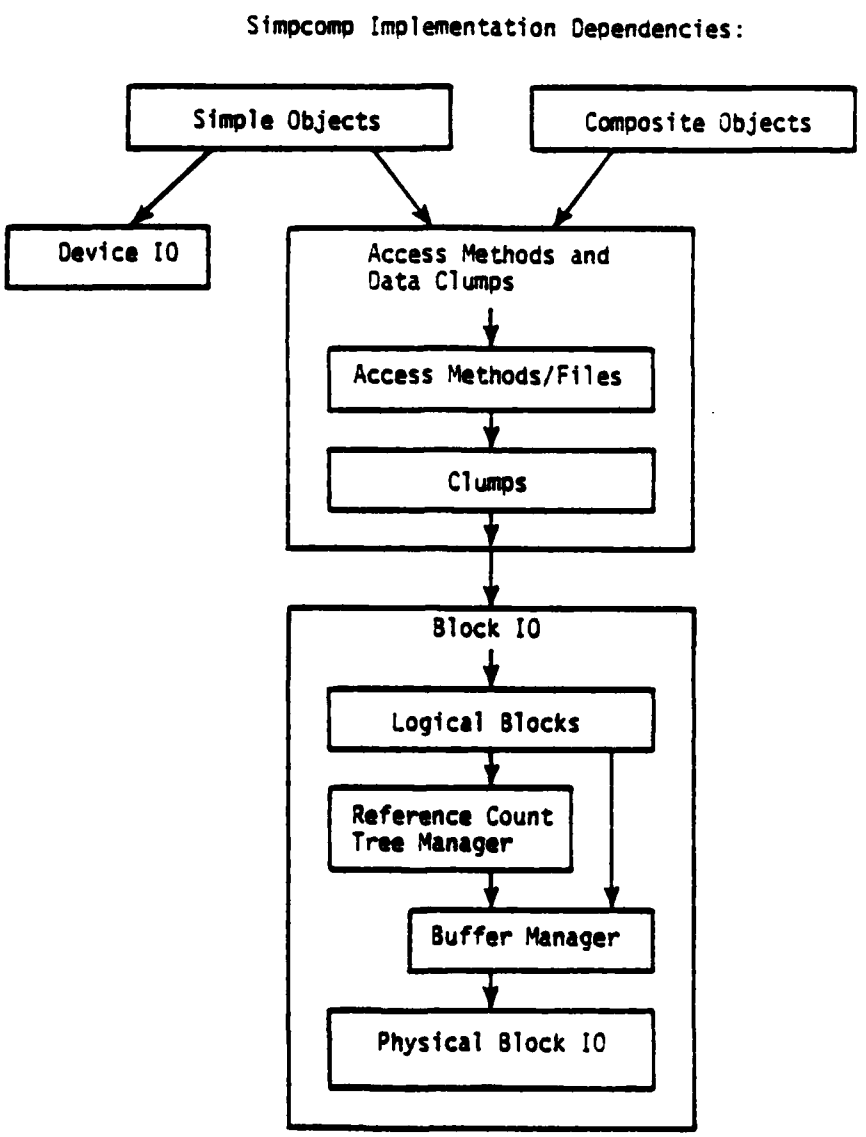
Top Level Implementation Dependencies:



3.2.5.1.3 ACCESS METHODS AND DATA CLUMPS

ACCESS METHODS AND DATA CLUMPS defines how file objects are implemented by the KAPSE. Data clumps are units of disk storage that are convenient for building access methods. Files are disk structures that are manipulated by access methods. The

Figure 3-9,
SIMPCOMP Inter-CPC Implementation Dependencies:



techniques used to implement the access methods are defined. The four different kinds of files implemented by the KAPSE are introduced:

a. Direct Access File;

b. Text Access File;

c. List Access File;

d. Key Access File.

3.2.5.1.4 SIMPLE OBJECTS

SIMPLE OBJECTS defines operations accessible at the KAPSE/Tool interface for manipulating extended objects with contents which are:

a. Direct Access File;

b. Text Access File.

3.2.5.1.5 COMPOSITE OBJECTS

COMPOSITE OBJECTS defines routines available as part of the KAPSE/Tool interface, for manipulating extended objects with contents which are:

a. List Access File;

b. Key Access File.

The routines to create composite database objects, to open and close partitions of existing composite objects, and to get the next component of a partition of a composite object are defined.

3.2.5.2 Access Control and Category (KAPSE.ACCECAT)

This CPCI defines the KAPSE's mechanism for controlling access to database objects, and the KAPSE's mechanism for classifying objects. The CPC's that comprise it are:

a. WINDOW OBJECTS

b. CATEGORY-DEFINED AND USER-DEFINED ATTRIBUTES

c. ACCESS CONTROL

Access control information in the KAPSE is distributed throughout the database. Each extended object contains information that defines the access rights available to users of the object, according to the user's "role." A role is an abstract property associated with a user that characterizes the activity

expected of the user (such as "PROGRAMMER", or "REVIEWER"). A role becomes associated with a user when the user goes through a window; going through windows is also the means by which a user traverses the database. Hence, the notions of access control, windows, and roles are all inter-related.

The KAPSE's CATEGORY facility is a mechanism whereby the user can define the structure and properties of a class of database objects. This facility is analogous to the concept of type in high level programming languages. An object's CATEGORY is stored as an attribute of the object. In general, an attribute of an object can itself be any kind of database object.

A brief discussion of each of the component CPC's follows.

3.2.5.2.1 WINDOW OBJECTS

WINDOW OBJECTS defines routines to create, delete, copy, and revoke windows.

3.2.5.2.2 CATEGORY-DEFINED AND USER-DEFINED ATTRIBUTES

This CPC defines routines:

a. to create and manipulate category descriptors;

b. to get and set the values of user-defined attributes.

A category descriptor is a list access file, each element of which describes the properties of a single category-defined attribute. The CATEGORY CPC defines routines to create and manipulate category descriptors, and thereby define new database object structures.

User-defined attributes are those attributes that are not defined by either the system or the category descriptor. Each extended object has a system-defined attribute called "USER_DEFINED_ATTRIBUTES" whose value is a keyed access file. The components of this attribute are keyed by user-defined attribute labels, strings that satisfy Ada naming conventions. The components have values that can in general be any kind of object.

When the user gives the label for an attribute, the KAPSE first checks to see if the label corresponds to a system-defined attribute; if not it then checks to see if it corresponds to a category-defined attribute of the object. If the label is not category-defined, it searches USER_DEFINED_ATTRIBUTES to see if it appears as a key to one of its components. Finally, if not there, the KAPSE repeats the process on the content of the object.

3.2.5.2.3 ACCESS CONTROL

ACCESS CONTROL provides routines to get and set the primitive access rights of an object that are associated with a particular role, as well as adjust the role translation table associated with a window.

3.2.5.3 Multiple Program Management (KAPSE.MULTPROG)

This CPCI defines how a program is invoked, how a program communicates with another program, and what the environment of a running program is. Also defined are the debugger interface to a running program and the KAPSE mail system.

The CPC's for this CPCI are:

a. PROGRAM LOADING

b. KAPSE PROGRAM COMMUNICATION

c. PROGRAM INVOCATION AND CONTROL

d. KAPSE KAPSE COMMUNICATION

e. TERMINAL SCREEN MANAGER

f. LOGIN/LOGOUT AND USER CONTEXT

g. MAIL

A brief discussion of each follows.

3.2.5.3.1 PROGRAM LOADING

PROGRAM LOADING defines the mechanism whereby programs are loaded into memory. It also defines the mechanism for sharing code between programs, and other host related issues involving needs of running programs. The PROGRAM LOADING CPC is part of the KAPSE/Host interface.

3.2.5.3.2 KAPSE PROGRAM COMMUNICATION

KAPSE PROGRAM COMMUNICATION defines the interface that allows a user program to request services of the KAPSE. To ensure integrity of the KAPSE, a protection boundary exists between the KAPSE and the user programs. The protection boundary is crossed only by bundling up a user request into a message and sending the message to the KAPSE via this special interface. This CPC forms part of the KAPSE/Host interface. interface.

3.2.5.3.3 PROGRAM INVOCATION AND CONTROL

PROGRAM INVOCATION AND CONTROL defines all interfaces associated with invoking a program, communicating with a running program, and manipulating a running program. In particular, routines are defined to do the following:

a. To call a program and wait for it to complete.

b. To initiate a program and not wait for it to complete.

c. To await the completion of a program.

d. To suspend and resume a program.

e. To invoke an operation defined for a database object.

f. To allow communication between running programs.

g. To debug a program.

3.2.5.3.4 KAPSE KAPSE COMMUNICATION

<<TBD>>

3.2.5.3.5 TERMINAL SCREEN MANAGER

TERMINAL SCREEN MANAGER defines an abstraction of an interactive terminal. This abstraction provides terminal control facilities to start and stop terminal output (XON, XOFF), to interrupt a running program (Control-C), to erase the previously typed character (Control-H), and to erase the current line (Control-X). It also implements a Scroll Control Mode that provides to the user commands to review text previously displayed on the terminal.

3.2.5.3.6 LOGIN/LOGOUT AND USER CONTEXT

LOGIN/LOGOUT AND USER CONTEXT provides routines to login, to obtain the user name of the currently executing user, to change the portion of the database viewed through the CURRENT_DATA window, and to change the user's password.

3.2.5.3.7 MAIL

MAIL provides routines to send a message, to check if a sent message has been read, to see if there are any waiting messages, and to read mail.

3.2.5.4 History and Archiving (KAPSE.HISTARCH)

This CPCI has three CPC's:

a. HISTORY

b. BACKUP RECOVERY

c. CONFIGURATION MANAGEMENT

3.2.5.4.1 HISTORY

HISTORY defines the mechanism whereby the KAPSE records enough information about an object to allow it to be reconstructed.

3.2.5.4.2 BACKUP RECOVERY

BACKUP RECOVERY defines routines to do a full backup (effectively a snapshot of the database), to do an incremental backup (recording only those blocks that have changed since the last backup), and to recreate a formerly backed up version of an object.

3.2.5.4.3 CONFIGURATION MANAGEMENT

CONFIGURATION MANAGEMENT defines a simple set of configuration management tools, as an example of the use of the configuration management support primitives provided within the KAPSE as part of other CPCs. The configuration management facility includes a tool to list the elements of a partition, as well as tools to reserve and release items of a configuration for the purpose of safe updating.

3.2.5.5 Run-time System (KAPSE.RTS)

This CPCI has the following CPC's:

a. UNIT EXECUTION SUPPORT

b. STORAGE MANAGEMENT

c. TASKING SUPPORT

d. EXCEPTION HANDLING

e. PREDEFINED PACKAGES

f. TYPE SUPPORT

A brief discussion of each follows:

3.2.5.5.1 UNIT EXECUTION SUPPORT

UNIT EXECUTION SUPPORT provides the basic support for the execution of sequential program units, including subprograms, blocks, and packages, and for the creation of local stack frames with a header, local variables, and subprogram communication area.

3.2.5.5.2 STORAGE MANAGEMENT

STORAGE MANAGEMENT provides routines for the allocation and management of the various run-time storage structures, including primary and secondary stacks, and controlled and checkpointed heaps.

3.2.5.5.3 TASKING SUPPORT

TASKING SUPPORT provides the basic routines for task creation, activation, synchronization, and termination.

3.2.5.5.4 EXCEPTION HANDLING

EXCEPTION HANDLING provides the basic support for raising and handling exceptions.

3.2.5.5.5 PREDEFINED PACKAGES

PREDEFINED PACKAGES implements the five predefined packages:

a. IO_EXCEPTIONS

b. SEQUENTIAL_IO

c. DIRECT_IO

d. TEXT_IO

e. CALENDAR

3.2.5.5.6 TYPE SUPPORT

TYPE SUPPORT provides routines to support basic operations on typed objects, such as fixed point arithmetic, and IMAGE and VALUE processing for scalar types.

3.3 Detailed Functional Requirements

3.3.1 Simple and Composite Objects (KAPSE.SIMPCOMP)

3.3.1.1 Block IO

3.3.1.1.1 Physical Disk I/O

3.3.1.1.1.1 Inputs and Outputs

The following low-level subprograms are implemented for each host, to provide physical disk I/O, as part of the KAPSE/Host interface:

```
with DATABASE_DEFS; use DATABASE_DEFS;  -- Host dependent
Package PHYSICAL_BLOCK_IO is

type BLOCK_ARRAY is array(1..BLOCK_SIZE) of STORAGE_UNIT;

procedure READ_BLOCK(BLK: in BLOCK_ID; DATA: out BLOCK_ARRAY);
        -- This procedure translates BLK into a physical
        --  disk address and then reads the block at
        --  that disk address into the buffer designated
        --  by DATA.  An entire block's contents is read.

procedure WRITE_BLOCK(BLK: in BLOCK_ID; DATA: in BLOCK_ARRAY);
        -- This procedure translates BLK into a physical
        --  disk address and then writes the contents of
        --  the buffer designated by DATA into that disk
        --  address.  An entire block's contents is
        --  written.

end PHYSICAL_BLOCK_IO;
```

3.3.1.1.1.2 Processing for VM/SP

Each KAPSE is given its own virtual machine which in turn is assigned a number of virtual mini-disks within the VM/SP Directory. Each of these mini-disks consist of a number of cylinders, with each cylinder holding a number of the KAPSE fixed-size blocks. A KAPSE data base can be logically viewed as an array of physical disk blocks, each block identified by a

unique block identifier (BLOCK_ID). Block identifiers are just integers. A separate function is provided to map Block identifiers into physical disk addresses (mini-disk, cylinder, track, byte).

Blocks are allocated so that sequential blocks are in the same cylinder, if possible, with a separation from the predecessor block determined by the physical characteristics of the device type of the mini-disk. The logically sequential blocks of an object are allocated non-contiguously to allow for the delays associated with a time-sharing environment, which prevent a user program from processing data as fast as the disk could provide it.

3.3.1.1.1.3 Processing for PE OS/32

The OS/32 KAPSE task obtains disk storage by creating contiguous OS/32 files with a consistent naming scheme. The files are then assigned to the KAPSE with exclusive read/write, thereby preventing other OS/32 tasks from corrupting the data. After creating such a file, it is treated much like the VM/SP mini-disk.

3.3.1.1.1.4 Special Requirements

The translation from BLOCK_ID to a physical disk address must be very efficient. Note that the only direct user of PHYSICAL_BLOCK_IO is the buffer manager (discussed below).

3.3.1.1.2 Buffer Management

3.3.1.1.2.1 Inputs and Outputs

The parameters to buffer management routines are typically a block identifier, a string to be read or written, and some "buffer management advice." Routines are also provided to flush a buffer, given the block identifier for the contents of the buffer, and to flush all the buffers.

Buffer management advice is information that the caller provides (as an in parameter) to the buffer management routines to help the buffer manager decide what to keep in memory and what to flush to disk. Buffer management advice is specified as a value from the enumeration: (WILL_NEED_AGAIN, WILL_NOT_NEED_AGAIN, NO_ADVICE).

3.3.1.1.2.2 Processing

Buffer management provides an in memory cache of the contents of recently referenced disk blocks. Its job is to minimize traffic to and from the disk, consistent with the legitimate needs of the caller (such as the periodic need of some applications to flush a particular buffer to disk).

Buffer management uses a "clock" algorithm in which all buffers being used are linked into a doubly linked circular list. This list (hereafter called the ring) is the "clock face" of the clock algorithm. A separate variable points to a particular element on the ring; this variable is the "hand" of the clock algorithm. When a buffer is needed, the hand sweeps forward around the ring looking for an element that indicates that it has not been recently referenced. As the hand passes over an element, it sets the referenced flag to "NOT_REFERENCED". In this way, even if it has to sweep the entire ring, the hand will eventually find a buffer with NOT_REFERENCED set. This algorithm implements an efficient approximation to "Least Recently Used."

Buffer management also maintains a free pool of buffers that have been removed from the ring. A threshold value determines when the pool must be replenished (via the algorithm described above).

The buffer manager advice allows additonal control over this caching algorithm, by causing it to favor or disfavor certain blocks by moving them in the ring.

The flush routine forces the contents of modified buffers to the disk, allowing higher level data base management routines control over the order in which physical writes occur. This is especially important for reduncancy/integrity structures such as transaction logs. The contents of a transaction log entry must be physically written to disk before the change it describes can be written to the disk.

The buffer manager uses physical block I/O. The only direct users of the buffer manager are the logical block manager and the reference count tree manager. Both are discussed below.

3.3.1.1.2.3 Special Requirements

Since all traffic to and from the disk is via the buffer manager, its performance is a very important part of the overall performance of the KAPSE. The algorithm will be tuned to provide a satisfactory level of performance.

3.3.1.1.3 Reference Count Tree

3.3.1.1.3.1 Inputs and Outputs

The primary input to the reference count tree manager is a block identifier. Routines are provided to get the reference count of a block, to increment it, and to decrement it; each of these return the reference count of the specified block as an out parameter. The routine to allocate a block returns the block id of the allocated block.

3.3.1.1.3.2 Processing

Reference counts are kept for every disk block in a database. The reference count of a block is the number of disk block pointers that point at the given block. If the reference count of a block is zero, then the block is free and can be allocated.

Reference counts are kept in a separate data structure called the reference count tree. The reference count tree has a root, some number of internal blocks, and leaf blocks. The root and all of the internal blocks contain nothing but disk pointers. The leaf blocks contain nothing but reference counts. The block id associated with a particular reference count is known implicitly by the position of the reference count in the reference count tree. The depth of the tree is uniform and is expected to be small.

3.3.1.1.3.3 Special Requirements

As with the buffer manager, this routine is critical for the overall performance of the KAPSE.

3.3.1.1.4 Logical Blocks

3.3.1.1.4.1 Inputs and Outputs

Logical blocks are identified by a limited private type, LOGICAL_BLOCK_HANDLE. A logical block handle is merely an access value on a data structure defined in the logical block manager package. The operations provided are to read and write the contents of a logical block, to copy all or part of a logical block's contents, to move all or part of a logical block's contents, and to allocate logical blocks.

The primary input is a logical block handle; secondary inputs specify what part of a logical block is to be moved or copied, or provide the data for write operations.

The outputs of logical block operations are logical block handles or the data that were read.

3.3.1.1.4.2 Processing

The content and attributes of all non-device (see 3.3.1.2) objects are recorded on the disk provided by the host machine. At a low level, all recorded information consists of either bytes of data or pointers to other disk blocks. In the KAPSE, this distinction between data bytes and disk pointers is made by the logical block manager.

Logical blocks are a logical view of a physical disk block. They contain two distinct parts: a data byte part and a block pointer part. Access to the data part of a logical block is unrestricted. Access to the block pointer part of a logical block, however, is restricted to a few operations, and the actual value of a block pointer is never revealed to the caller. The caller must identify a block pointer by its location in the block pointer section. The location of a block pointer is referred to as a "slot," and the caller refers to a slot by a "slot number." Access to block pointers is controlled to maintain for every disk block an accurate count of all references to that block, needed for "virtual" copying (see below).

Operations on the byte portion of a logical block are to read or write all or part of the bytes stored.

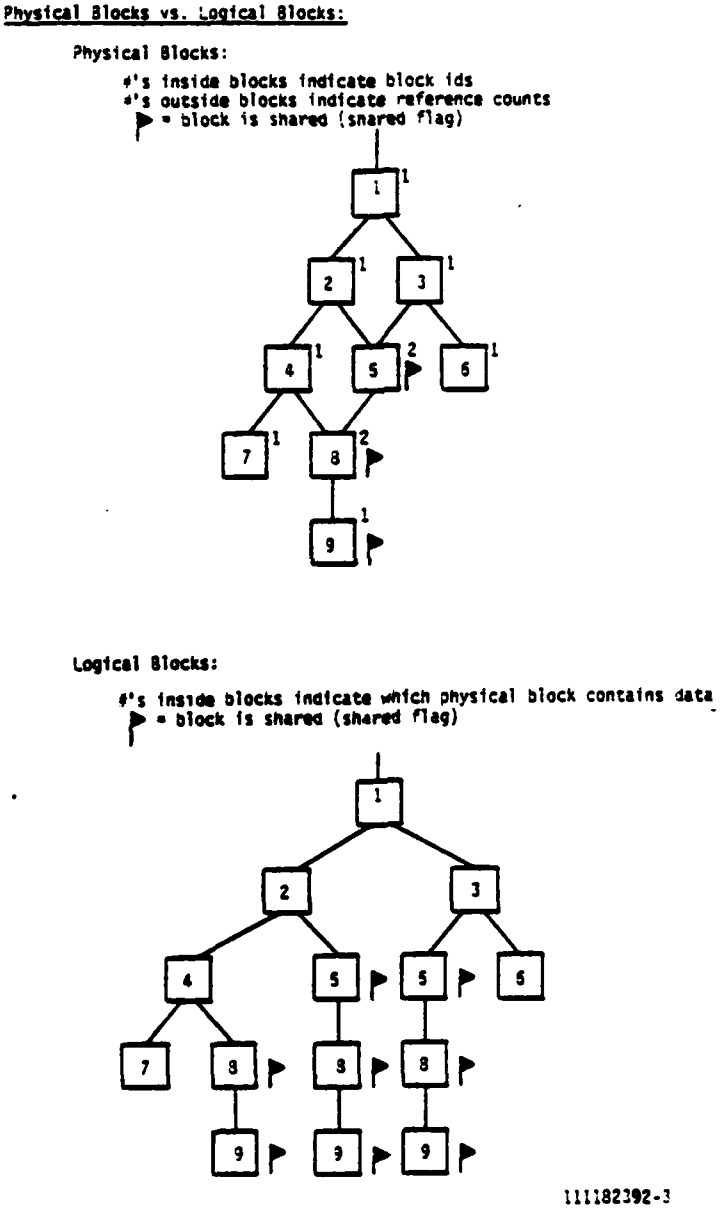
Operations on the block pointer portion of a logical block are to allocate a new block and store its identifier in a given slot, to erase the block pointer stored in a given slot, to copy a pointer from slot from one slot to another, and to move a pointer slot from one slot to another.

The graph formed by disk block pointers is guaranteed by the logical block manager to be acyclic, and there is a well-defined root for the entire graph. This implies that the "children" of a block (those pointed to from one of the block's slots) can never be one of its "ancestors." Combined with the fact that the reference count of each disk block in the database is known, this makes it possible to determine if a block is shared. A block is shared if its reference count is greater than one, or if any of its ancestors along any path from the root are shared.

By keeping track of whether a block is shared, physical copying of the block can be deferred until a change is actually made to the original or some logical copy. To logically copy a block and all of its decendants, one need only increment the

reference count of the top block (making it and all of its descendants shared); one need not physically copy all the specified blocks. This mechanism of deferring actual copying is referred to as "virtual" copying. See Figure 3-10.

Figure 3-10, Physical Blocks vs. Logical Blocks:



When a logical block is to be changed, and the logical block manager determines that it is in fact shared (by keeping track of whether it or any of its ancestors have multiple references), then a new block must be allocated to hold the changed data. Since the changed data will be stored in a new disk block, the parent of the old block (on the path taken from the root to this block) must be changed to point to the new block. As before we must check to see if the parent is shared to see if we can change the parent in place or if we need to allocate another disk block. Eventually we will find a parent that is not shared, and can be updated in place.

Given an unshared parent block, we allocate a new disk block and copy the contents of the old child of the parent into the newly allocated block (incrementing the reference counts of all blocks referred to by slots in the child, and setting the initial reference count for the new block is set to be one). The parent is then changed in place to point (via the appropriate slot) to the new block and the reference count of the old child is decremented. At this point the new child contains an exact copy of the contents of the old child, but unlike the old child it is not shared. Hence, it can be changed in place and the process repeats (with the new child acting as the parent), until we reach the original block that was to be changed.

The eventual amount of physical block copying required with the logical copy approach is never more than if the object were physically copied immediately, and is generally significantly less for large, relatively stable objects. The cost of maintaining reference counts, however, can affect overall system performance.

The header of a logical block also records the TIME_SEQUENCE_NUMBER when the block was last written, which is useful for system checkpoint and incremental backup (see 3.3.4.2).

3.3.1.1.4.3 Special Requirements

The virtual copy mechanism is central to the design of many of the KAPSE's features. It permits multiple copies of stable objects to be very space-efficient. In this way, for instance, a category descriptor may be logically contained in thousands of objects, but only occupy the space required for one copy. The virtual copy mechanism is relied upon to efficiently store many versions of the same object. It is also used to implement the synchronization access modes (WRITE_COPY, WRITE_ORIGINAL), by making a copy of the object that can be manipulated safely (see 3.3.2.3).

It is essential, therefore, that the virtual copy mechanism be reliable and efficient.

The logical block manager uses both the buffer manager and the reference count tree manager. It in turn is used only by the clump manager (see data clumps and access methods).

3.3.1.2 Device IO

A small number of device objects are created by the system manager to provide direct and import/export access to physical I/O devices or disk files of the host system.

3.3.1.2.1 Terminal I/O

3.3.1.2.1.1 Inputs and Outputs

The following primitives are available to the KAPSE for terminal input/output:

```
   Package TERMINAL_IO is

   procedure READ_TERMINAL(TERM: in INTEGER; ECHO: in BOOLEAN;
       DATA: in BUFFER_PTR);
           -- This procedure sets up a buffer for characters
           --  to be read from the specified terminal,
           --  with or without echoing.
           -- The buffer will be released when full, or when any
           --  ASCII control character is typed (including DEL).
           -- NUM_BITS of the associated BUFFER_DATA
           --  indicates actual number of characters accepted.
           -- With MAX_NUM_BITS => ASCII.CHARACTER'SIZE,
           --  the buffer is filled as soon as
           --  the next character is typed.
           -- ASCII control characters are never echoed
           --  by READ_TERMINAL, independent of ECHO.

   procedure WRITE_TERMINAL(TERM: in INTEGER;
       DATA: in BUFFER_PTR)
           -- This procedure writes characters to the
           --  specified terminal.
           -- DATA must have been filled in previously,
           --  and will be drained asynchronously.

   procedure SET_TERMINAL_INFO(TERM: in INTEGER;
       INFO: in TERMINAL_INFO_BLOCK);
   procedure GET_TERMINAL_INFO(TERM: in INTEGER;
       INFO: out TERMINAL_INFO_BLOCK);
           -- These procedures pass along information
           --  between the host terminal device driver
           --  and the KAPSE terminal handler.
           -- In the case of hard-wired terminals, the host
           --  may know the characteristics of the
           --  terminal.  For dial-up terminals, the user
           --  must in general specify the appropriate
           --  information explicitly via SET_INPUT_INFO
           --  and SET_OUTPUT_INFO (see *** above),
           --  which the KAPSE will then digest and send
           --  along via SET_TERMINAL_INFO.

   end TERMINAL_IO;
```

3.3.1.2.1.2 Processing for VM/SP

3.3.1.2.1.3 Processing for PE OS/32

The KAPSE task on OS/32 handles all terminal I/O for the KAPSE. Individual user tasks need not be rolled in for echoing to

proceed, and character and line deletion to be processed.

For each user a separate Ada task within the KAPSE handles the terminal. When an input buffer is complete, the waiting user program OS/32 task is activated by sending it a message containing the characters.

3.3.1.2.1.4 Special Requirements

3.3.1.2.2 Other Device Input/Output and Import/Export

3.3.1.2.2.1 Inputs and Outputs

Device objects (see CREATE_DEVICE_OBJ above) are used as the access points for device I/O and Import and export. Because only a system manager may create device objects, the correct syntax for HOST_DEVICE_NAME need not be known to the normal user, and may be host-dependent.

The following primitives exist for the KAPSE to read or write host files or physical I/O devices:

```
Package DEVICE_IO is

type FILE_MODE is (IN_MODE, INOUT_MODE, OUT_MODE);
type DEVICE_HANDLE is private;

OPEN_DEVICE(DH: in out DEVICE_HANDLE;
    HOST_DEVICE_NAME: in STRING; MODE: in FILE_MODE);

READ_DEVICE(DH: in DEVICE_HANDLE; DATA: in BUFFER_PTR);

WRITE_DEVICE(DH: in DEVICE_HANDLE; DATA: in BUFFER_PTR);

CLOSE_DEVICE(DH: in out DEVICE_HANDLE);
        -- Whenever a user reads or writes a device
        --  object, the KAPSE retrieves the HOST_DEVICE_NAME
        --  stored when the device object was created,
        --  and passes the request off to these KAPSE/Host
        --  interface procedures.

SET_DEVICE_INFO(DH: in DEVICE_HANDLE;
    INFO: in DEVICE_INFO_BLOCK);

GET_DEVICE_INFO(DH: in DEVICE_HANDLE;
    INFO: out DEVICE_INFO_BLOCK);
        -- A certain amount of device control and status
        --  information may be set and retrieved using
        --  these calls.  These are externally accessible
        --  as KAPSE calls SET_FILE_INFO and GET_FILE_INFO.

end DEVICE_IO;
```

3.3.1.2.2.2 Processing for VM/SP

On the VM/SP the HOST_DEVICE_NAME implies the virtual device address and device type. Using commands to VM/SP CP, a user or operator can connect what appears to be a virtual punch on one VM to be a virtual card reader on some other VM. In this way, export/import can be with actual devices, or files on other operating systems.

3.3.1.2.2.3 Processing for PE OS/32

On OS/32 the HOST_DEVICE_NAME implies the physical device mnemonic, or the volume and file name of the host file.

3.3.1.2.2.4 Special Requirements

3.3.1.3 Access Methods and Data Clumps

3.3.1.3.1 Data Clumps

3.3.1.3.1.1 Inputs

The routines of the data clump package within the KAPSE take as input a specification of the clump to be created, read, written, or otherwise manipulated, in general in the form of a handle on the clump or one of its neighbors. When a simple clump is being written, the input includes the data to write into the clump. When a clump is being created, the input includes a specification of the "size" (block/sub) and "kind" (simple/composite) of the clump.

3.3.1.3.1.2 Processing

Recognizing the desireability of objects smaller than a disk block, data clumps (or just "clumps") implement disk storage units which can have any size up to a maximum equal to the size of a disk block. Clumps are built on top of logical blocks and like logical blocks consist of a data byte portion and a slot portion. A logical block may be divided up into many clumps which are constrained to form a well structured hierarchy (described below). The bytes and slots of a logical block that belong to a given clump are determined from that clump's byte_offset, byte_count, slot_offset, and slot_count fields.

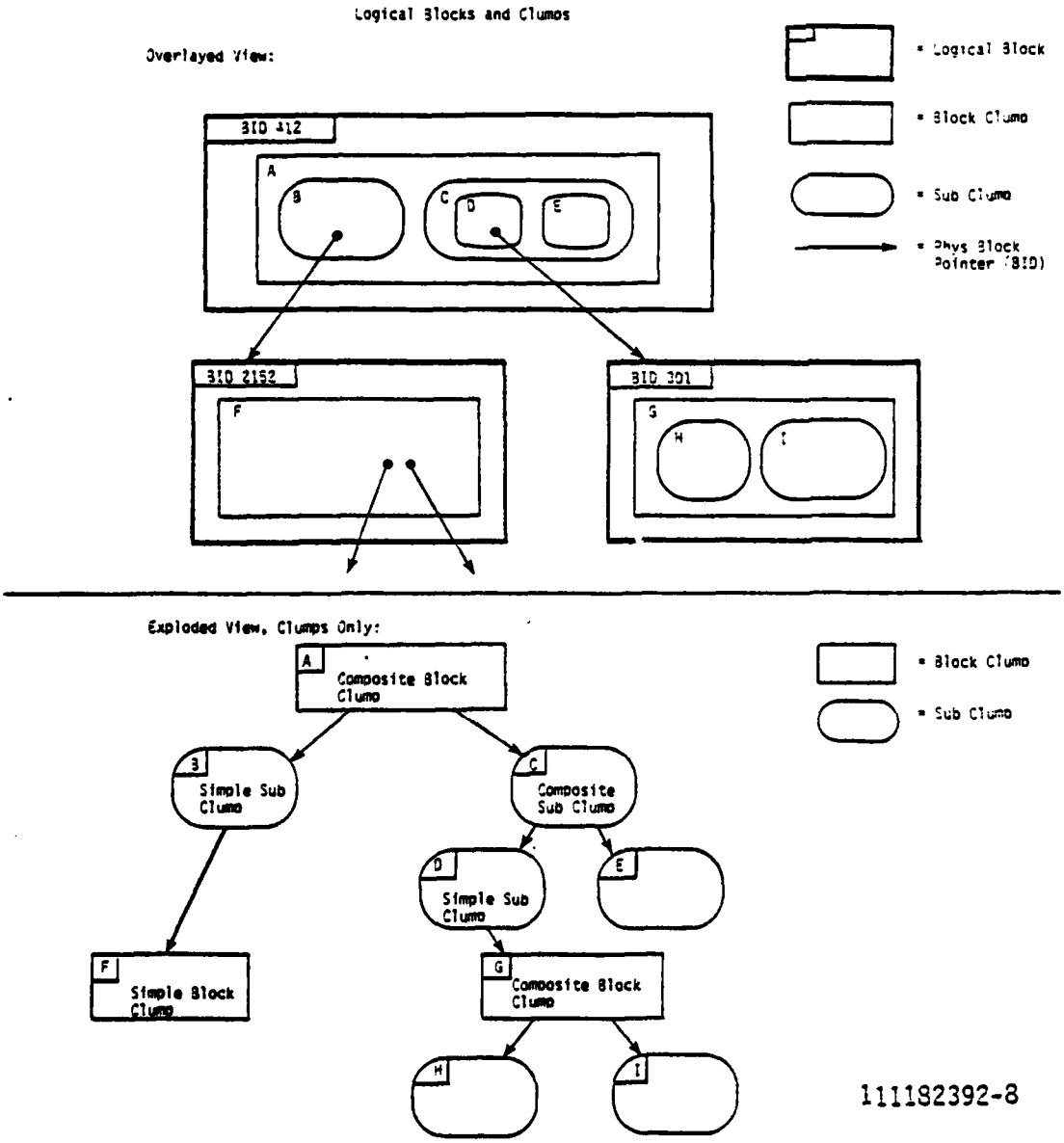
Clumps are further categorized by their constituents: "Composite" clumps are those whose contents consist of smaller clumps. "Simple" clumps are those whose contents are interpreted directly as a group of bytes and slots. An entire logical block may be thought of as a clump, with sub-clumps if it is "composite," or bytes and slots if it is "simple." This top-level clump will be called a "block" clump, while others are called "sub"-clumps, or simply "clumps." See Figure 3-11. All four combinations of block vs. sub, and simple vs. composite are possible:

a. Simple Block Clump -- Basically just a "logical block" under a fancier name.

b. Composite Block Clump -- A logical block made up of sub-clumps.

c. Simple Sub-Clump -- A constituent of some composite clump, whose data is directly interpreted as bytes and slots.

Figure 3-11, Logical Blocks and Clumps:



d. **Composite Sub-Clump** -- A constituent of some composite clump, made up of further sub-clumps.

The byte and slot counts of a clump are actually stored with it in the block. The offsets are derived during in-memory

processing. Clumps are normally processed sequentially from left to right within a block, and therefore the slot offset may be found by counting the number of slots allocated to all clumps to its "left" in the block.

3.3.1.3.1.3 Outputs

The routines of the clump package return as output the handle on a created or located clump, as well the contents of a simple clump when it is being read. Some routines also report the size/kind of the clump.

3.3.1.3.1.4 Special Requirements

The maximum size of a block clump is determined by the block size chosen for the database as a whole. This is allowed to be host- and medium-dependent for efficiency. No user of clumps should rely on the exact maximum size of block clumps. It is required that this maximum be at least 500 bytes for all databases, and this lower limit may be safely depended-on, independent of the host.

Sub-clumps are purposely limited to a size that is less than any anticipated database block size, to make the limitation host-independent. Sub-clumps are designed for rapid left-to-right processing, and proper use of them requires taking advantage of this design.

3.3.1.3.2 Access Methods

3.3.1.3.2.1 Inputs

The routines of the access method packages generally take an identification of the file being manipulated, in the form of a basic object handle, or a component specifier (parent plus selector). If the operation is a write to a simple file, the data to be written is also an input. If the operation is a create, a specification of the access method for the file is an input.

3.3.1.3.2.2 Processing

Data clumps are not normally visible at the user level. Instead, all data is organized into primitive data files, each primitive data file managed by some "access method" which provides for their creation, expansion, modification, interrogation, compression, and deletion.

Access methods use clumps for all data storage purposes. When a data file is small, a single (composite or simple) clump is sufficient to represent it. This kind of file is called an "embedded" file. When a data file grows too large to fit in a single clump, the access method allocates additional block clumps and manages them in some kind of multi-way tree structure. This kind of file is called a "multi-block" file. Multi-block files have their data spread across a file header clump, a set of leaf block clumps, and sufficient "internal" block clumps to provide efficient and complete access to the leaf block clumps.

When a particular file offset is requested, the multi-way tree structure is walked by the access method, starting at the top, following down the branch figured to contain the desired data. The number of disk block references on average is equal to the height of the tree. The height of the tree is kept low by ensuring that each block is at least half full of data, using a variant of the well-known B*-tree mechanism [Knuth73]. With an average branching factor of BF in each block, and a total of N leaf blocks, the height will be approximately (log N/log BF).

Several different access methods are supported by the KAPSE:

a. Direct Access Method -- This provides to the user program an arbitrarily extendable file of bytes, indexed by byte position, with a user adjustable first- and last-defined byte position.

   By adding bytes to the end of the file, and removing bytes from the beginning of the file (i.e. advancing the first-defined byte position), a direct-access file can be used as a FIFO stream of bytes.

   Even short Ada program objects may be efficiently stored using the direct access method, with the entire object in a single simple clump (see "simple clumps" above).

b. Text Access Method -- This provides to the user program an arbitrarily extendable and editable file of ASCII text, indexed by both character position, and line number. The user may insert and delete characters and lines anywhere in the file. A single ASCII character, the standard "line-feed" character is used as a line separator within the file.

   By adding characters to the end of the file, and deleting characters at the beginning of the file, a text file may be used as a FIFO stream of characters (or lines).

   Short ASCII strings are represented using the text access method, with the entire file held in a single simple clump (see "simple clumps" above).

c. Key Access Method -- This provides to the user program a primitive composite file, whose components are objects identified by an ASCII string key. The key may be as short as one ASCII character, or as long as 100 ASCII characters. The components may be any kinds of files. The internals of the components are managed by their respective access methods.

d. List Access Method -- This provides to the user program a primitive composite file, whose components are objects indexed by list position. The list of objects is arbitrarily extendable, and the list positions are numbered from one to the number of objects in the list. As with the Key Access Method, the internals of the components are managed by their own access methods.

Deletion of objects from the front of the list provides a kind of object FIFO queue.

Short list files will be represented in a single composite clump, with no additional indirect blocks necessary.

e. Extended Access Method -- This provides to the user program an extended, higher level object, with the structure, kinds of attributes, and defined operations controlled by its "category descriptor." This is not really a new access method, but rather built on top of the above four access methods.

Internally, an extended object appears as a composite "list" file (see above), with the first element of the list by convention being the category descriptor, and the remaining elements being used for the content and other attributes of the object, as specified by the descriptor.

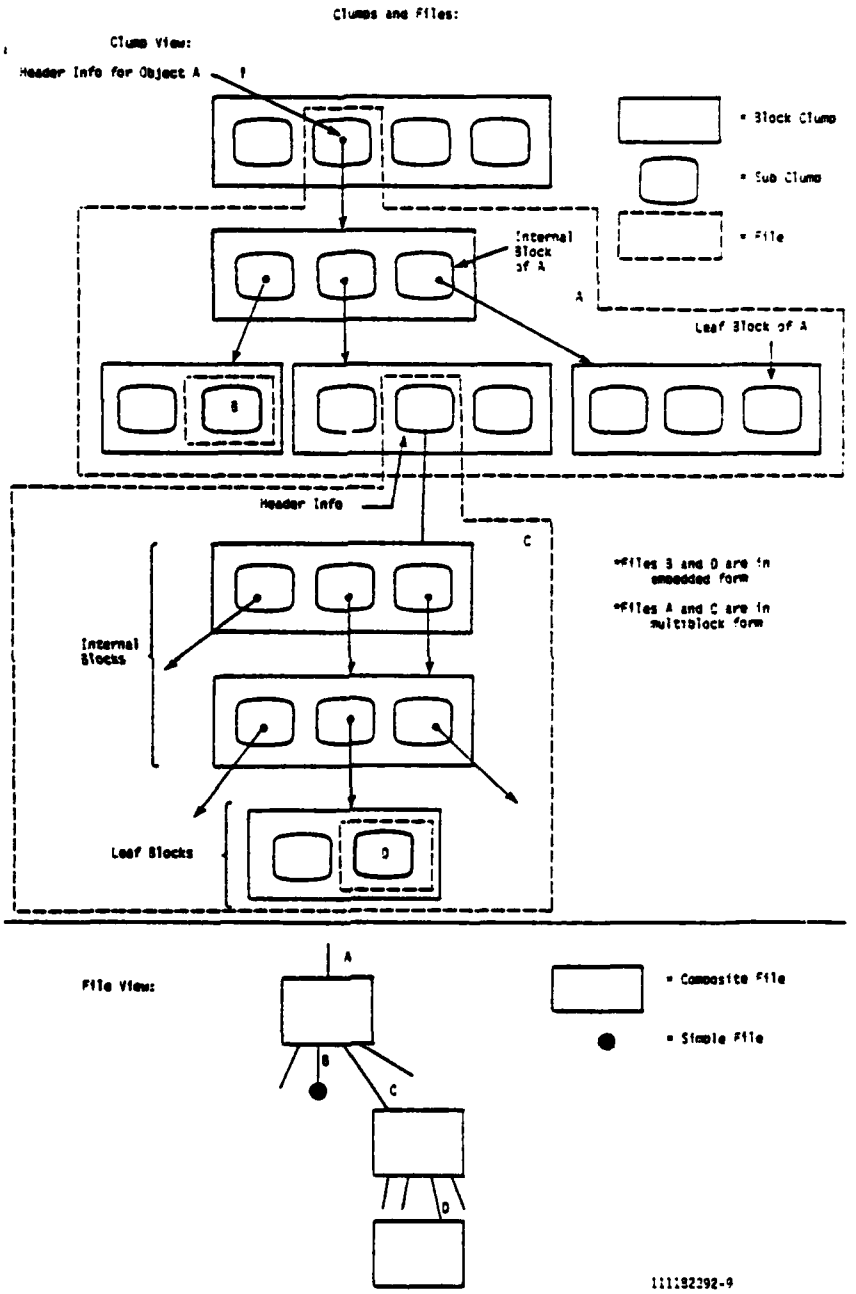
A clump used to represent an entire file, or an file header, begins with an "access method code," which specifies it as being managed by one of the above access methods. Block clumps also begin with an "access method code," which specifies which access method is responsible for handling block overflow.

Each access method is responsible for distinguishing files in embedded single-clump form from those in multi-block form (with a header, internal blocks, and leaf blocks). See Figure 3-12.

3.3.1.3.2.3 Outputs

The outputs of the routines of the access method packages include a basic object handle on the file if it is being created, or the data from the file if the operation is a simple file "read."

Figure 3-12, Clumps and Files:



3.3.1.3.2.4 Special Requirements

The performance of the access methods are critical to the performance of the KAPSE database as a whole. The access methods must be designed to minimize the numbers of blocks accessed to locate the desired element of a file, as well as minimize the number of blocks affected when an element is added or removed from the file.

All access methods are based on a B*-tree [Knuth73] structure, which provides the desirable logarithmic dependency on the number of elements. The other major determinant of B*-tree performance is occupancy of the blocks, with both time and space performance being improved when the blocks are more nearly filled. The access methods must be designed to maximize occupancy of the blocks consistent with the requirements of efficient addition and removal of elements for typical access patterns.

Finally, the "embedded" form of files is essential to the efficient storage of small objects within the database. The algorithms within the access methods for determining when to go to multi-block form must be carefully designed to minimize internal fragmentation caused by small objects ending up being allocated an entire block.

3.3.1.4 Simple Objects

3.3.1.4.1 Inputs and Outputs

The primary user-visible interface to simple objects is provided by the set of standard Ada input/output packages specified in the [LRM82, 14.1]. These packages are implemented in terms of a more primitive set of access methods. See 3.3.5.5 for a definition of the package IO_EXCEPTIONS and the skeletons for the other language-defined I/O packages.

The basic primitives available are as follows:

```
with IO_COMMON; use IO_COMMON;
Package SIMPLE_OBJECTS is

  procedure COPY(OLDNAME: in STRING; NEWNAME: in STRING);
        -- This procedure creates a logical copy of the
        --  specified object, with identical content and
        --  non-distinguishing attributes.  The
        --  distinguishing attributes of the copy are
        --  implied by NEWNAME.
        -- COPY involves no actual disk data block copying.
        --  When either the original or copy is later
        --  modified, the KAPSE makes actual physical
        --  copies of the affected blocks.

  procedure DELETE(NAME: in STRING);
        -- Requires DELETE_COMPONENT access on
        --  the enclosing composite object.

  procedure RENAME(OLDNAME: in STRING; NEWNAME: in STRING);
        -- Defined to be equivalent to COPY followed by DELETE of
        --  OLDNAME.
```

```
Package DIRECT_ACCESS is
      -- Package to provide direct access file (a sequence
      --  of storage units).

  type FILE_TYPE is private;
  subtype STORAGE_UNIT is Machine_Dependent;
  type STORAGE_ARRAY is array(NATURAL range <>) of STORAGE_UNIT;
      -- All data converted to/from an array of storage units.

  procedure CREATE(FILE: in out FILE_TYPE; MODE: in FILE_MODE;
      NAME: in STRING; FORM: in STRING := "");
  procedure OPEN  (FILE: in out FILE_TYPE; MODE: in FILE_MODE;
      NAME: in STRING; FORM: in STRING := "");
  procedure CLOSE (FILE: in out FILE_TYPE);
      -- Create/open/close designated simple object.
      -- The MODE selects input only, output only, or inout.
      -- The NAME is a pathname to the object.
      -- The FORM is an optional parameter, which
      --  supplies additional control info (see below)

  procedure WRITE (FILE: in FILE_TYPE; ITEM: in  STORAGE_ARRAY);
  procedure READ  (FILE: in FILE_TYPE; ITEM: out STORAGE_ARRAY;
      LAST: out COUNT);
      -- ITEM'length determines amount to READ/WRITE.
      -- LAST specifies amount actually READ if reached end
      --  of file (ITEM(LAST) is last valid data).

  procedure SET_OFFSET(FILE: in FILE_TYPE; TO: in COUNT);
  function  OFFSET    (FILE: in FILE_TYPE) return COUNT;
      -- SET_OFFSET selects which storage unit within file
      --  to read/write next.
      -- OFFSET returns offset of storage unit to be
      --  next read/written.

  function SIZE(FILE: in FILE_TYPE) return COUNT;
      -- return count of storage units in file.
end DIRECT_ACCESS;
```

B5-AIE(1).KAPSE(1)

```
Package TEXT_ACCESS is
     -- Package to provide a text file (a sequence of
     --  ASCII characters, accessible by character or
     --  line).
  procedure CREATE(FILE: in out FILE_TYPE; MODE: in FILE_MODE;
     NAME: in STRING; FORM: in STRING := "");
  procedure OPEN  (FILE: in out FILE_TYPE; MODE: in FILE_MODE;
     NAME: in STRING; FORM: in STRING := "");
  procedure CLOSE (FILE: in out FILE_TYPE);

  procedure WRITE (FILE: in FILE_TYPE; ITEM: in  STRING);
  procedure READ  (FILE: in FILE_TYPE; ITEM: out STRING;
     LAST: out COUNT);
     -- ITEM'length determines amount to READ/WRITE.
     -- LAST specifies amount actually READ if reached end
     --  of file (ITEM(LAST) is last valid data).

  procedure READ_LINE(FILE: in FILE_TYPE; ITEM: out STRING;
     LAST: out COUNT);
     -- ITEM'length determines maximum amount to READ.
     -- LAST specifies amount actually READ if reached end
     --  of line (ITEM(LAST) is last valid data).


  procedure SET_OFFSET(FILE: in FILE_TYPE; TO: in COUNT);
  function  OFFSET    (FILE: in FILE_TYPE) return COUNT;
     -- SET_OFFSET selects which character within file
     --  to read/write next.
     -- OFFSET returns offset of character to be
     --  next read/written.

  procedure SET_LINE  (FILE: in FILE_TYPE; TO: in COUNT);
  function  LINE      (FILE: in FILE_TYPE) return COUNT;
     -- SET_LINE positions at beginning of selected line.
     --  The first line is always numbered 1.
     -- LINE returns current line's number.

  procedure SET_COL   (FILE: in FILE_TYPE; TO: in COUNT);
  function  COL       (FILE: in FILE_TYPE) return COUNT;
     -- SET_COL positions at given column within line.
     -- COL returns current column number within line.

  function CHAR_COUNT(FILE: in FILE_TYPE) return COUNT;
  function LINE_COUNT(FILE: in FILE_TYPE) return COUNT;
     -- return count of characters/lines in
     --  the text file.
end TEXT_ACCESS;
```

```
procedure CREATE_DEVICE_OBJ(NAME: in STRING;
    HOST_DEVICE_NAME: in STRING; ROOT_WINDOW: in STRING);
        -- This procedure is provided for a system
        --  manager to set up an association between
        --  a special database object and
        --  a host physical I/O device.  The
        --  HOST_DEVICE_NAME is host-dependent.
        -- Requires CREATE_COMPONENT access,
        --  as well as a .SYSTEM window on the root
        --  of the database (restricted to system manager).

end SIMPLE_OBJECTS;
```

3.3.1.4.2 Processing

The objects created by the packages of SIMPLE_OBJECTS are all simple "extended" objects, with a default null CATEGORY_DESCRIPTOR, and a CLASS of SIMPLE. The content of the extended object is a simple file, either a direct-access file or a text-access file (see access methods above).

Most of the processing within the SIMPLE_OBJECTS packages consists of calling the appropriate access method routines. However, the initial creation requires building the extended object using a list access file with pre-defined elements for the CATEGORY_DESCRIPTOR, the ACCESS_CONTROL, CONTENT, HISTORY, etc.

Opening an existing object requires the creation of an extended object handle, with the implicit "offset" initialized to the beginning of the file. The handle must be entered in the table of open file handles associated with the running program context object.

All operations in packages of the KAPSE/Tool interface must verify that the proper access controls are applied. The checking is performed by the access control CPC of KAPSE.ACCECAT (see 3.3.2) as part of its pathname lookup and handle initialization routines.

The FORM STRING passed to OPEN or CREATE may be used to convey extra information. The additional information is in the form of a label=>value list. With this syntax, it is possible to specify the following extra information:

Call	Extra labeled FORM specification
CREATE	RESERVE_MODE, ACCESS_CONTROL, CATEGORY_DESCRIPTOR, ACCESS_METHOD
OPEN	RESERVE_MODE

For example:

```
OPEN ( FILE1, "STREAM_OBJECT_1",  "RESERVE_MODE=>SHARED_STREAM" );
CREATE ( FILE2, "PUBLIC_INFO_FILE",
    "ACCESS_CONTROL=>(WORLD=>(READ,ADD))" );
```

3.3.1.4.3 Special Requirements

This package, because it is part of the KAPSE/Tool interface package, must ensure that the access control and synchronization requirements are met. Lower level packages (such as the access methods) assume that access control has been checked at the higher level.

3.3.1.4.4 Interactive/Terminal I/O Extensions

3.3.1.4.4.1 Inputs and Outputs

As an addition to the facilities of the standard Ada package TEXT_IO [LRM82, 14.3] (see 3.3.5.5), we make available the random access by line number or character number of the package SIMPLE_OBJECTS.TEXT_ACCESS (see above), and a package to handle echoing and special character processing:

```
with TEXT_IO; use TEXT_IO;
with TEXT_ACCESS; use TEXT_ACCESS;
Package INTERACTIVE_IO is

type FILE_TYPE is TEXT_IO.FILE_TYPE;

procedure SET_ECHO(INPUT: in FILE_TYPE; OUTPUT: in FILE_TYPE);
        -- Sets "cursor" and echoing of INPUT at current
        --  line and column of output.  Each character GET from
        --  INPUT advances the column of both the INPUT and
        --  the OUTPUT files (although the column numbers will
        --  not necessarily be the same).

procedure NO_ECHO(INPUT: in FILE_TYPE);
procedure NO_ECHO(OUTPUT: in FILE_TYPE);
        -- Either of these calls will break any
        --  echoing association.

procedure GET_OUTPUT_INFO(FILE: in FILE_TYPE;
    INFO: out OUTPUT_INFO_BLOCK);

procedure SET_OUTPUT_INFO(FILE: in FILE_TYPE;
    INFO: in OUTPUT_INFO_BLOCK);
        -- The OUTPUT_INFO_BLOCK retains information such as
        --  the terminal's screen height and width (zero height
        --  indicates hard copy, zero width indicates FILE_TYPE
        --  is not associated with a physical terminal).

procedure GET_INPUT_INFO(FILE: in FILE_TYPE;
    INFO: out INPUT_INFO_BLOCK);

procedure SET_INPUT_INFO(FILE: in FILE_TYPE;
    INFO: in INPUT_INFO_BLOCK);
        -- The INPUT_INFO_BLOCK retains information such as
        --  the specific keyboard control characters used to
        --  control the various terminal handling functions.
        -- In addition, the INPUT_INFO_BLOCK records
        --  which characters cause program wakeup when
        --  typed (others are buffered up and a control
        --  character may be used to delete them
        --  before they are received by a program).

end INTERACTIVE_IO;
```

3.3.1.4.4.2 Processing

All terminal output is actually written to a temporary file in the program's context object. The terminal handler normally keeps the last line of this temporary file as the last line on the screen. However, the user may choose to scroll backward to

see previous lines of output, or to simply hold the screen image at a particular line. When echoing is set, the terminal handler makes sure that the current LINE and COL of the output are on the screen before setting the cursor there and requesting input on the associated FILE_TYPE.

3.3.1.4.4.3 Special Requirements

The SET_ECHO routines must work on terminals with local hardware echo, full-duplex terminals without local echo, and normal text files. In all cases, the effects should be meaningful and analogous.

3.3.1.4.5 Package FORMATTED_IO

3.3.1.4.5.1 Inputs and Outputs

Along with the above adjunct to TEXT_IO, the KAPSE defines a FORMATTED_IO package to provide the facilities of Fortran-like FORMAT I/O:

```
with TEXT_IO;
Package FORMATTED_IO is

type FORMAT is private;

function CONV_FMT(FMT: in STRING) return FORMAT;
        -- Given a STRING in Fortran FORMAT syntax, check
        --  the correctness of the syntax and compress to
        --  facilitate further use.

procedure FWRITE(FILE: in TEXT_IO.FILE_TYPE; FMT: in FORMAT);
        -- Start output using the given (compressed) FORMAT.

procedure FPUT(ITEM: in STRING);
        -- This uses the "Aw" format.
procedure FPUT(ITEM: in FLOAT);
        -- This typically uses "Fw.d" formats.
procedure FPUT(ITEM: in INTEGER);
        -- This typically uses the "Iw" format.
        -- Continue output, using the next format specifier
        --  from the format specified in the most recent FWRITE call.
        -- The user may choose to further overload FPUT by writing
        --  versions that take a sequence of INTEGERS or FLOATS or
        --  some useful combination.

procedure FEND;
        -- Terminate output, force characters out to file.

procedure FREAD(FILE: in TEXT_IO.FILE_TYPE; FMT: in FORMAT);
        -- Start input using the given (compressed) FORMAT.

procedure FGET(ITEM: out FLOAT);
        -- This typically uses the "Fw.d" format.
procedure FGET(ITEM: out INTEGER);
        -- This typically uses the "Iw" format.
        -- Continue input, using the next format specifier from
        --  the FORMAT specified in the most recent FREAD call.
        -- The user may choose to further overload FGET by writing
        --  versions that take a sequence of INTEGERS or FLOATS
        --  or some other useful combination.

 ...

end FORMATTED_IO;
```

3.3.1.4.5.2 Processing

The package FORMATTED_IO is implemented in Ada, using package TEXT_IO and package INPUT_OUTPUT, ensuring that it is easily transportable to other Ada installations.

3.3.1.4.5.3 Examples

```
declare
        F1: constant FORMAT := CONV_FMT( "2I3, F8.2" );
        I,J,K: INTEGER := 5;
        Z: FLOAT := 3.22;
begin
        FWRITE(FILE, F1)
        FPUT(I+J); FPUT(25); FPUT(Z); FEND;

        FWRITE(FILE,CONV_FMT(" 'The Answer is ',I6//"));
        FPUT(K*127); FEND;
end;
```

3.3.1.4.5.4 Special Requirements

The package FORMATTED_IO need not be within the protection boundary surrounding the KAPSE, and hence the actual body of the FORMATTED_IO package may be linked directly into user Ada programs.

3.3.1.5 Composite Objects

3.3.1.5.1 Inputs and Outputs

The following primitives are available for creating and modifying composite objects:

```
Package COMPOSITE_OBJECTS is

procedure CREATE_COMPOSITE(NAME: in STRING; COMPONENT_DA: in STRING;
    FORM: in STRING := "");
        -- COMPONENT_DA is a space separated list of attribute
        --  labels required of all components created in the object.
        -- Requires CREATE_COMPONENT access on the enclosing
        --  composite object.
        -- FORM is used to supply additional category description

type PARTITION_HANDLE is private; -- Similar to FILE_HANDLE.

procedure OPEN_PARTITION(PH: in out PARTITION_HANDLE; NAME: in STRING);
        -- NAME is a specification of a partition,
        --  like "(PROJECT=>SHUTTLE)" or "*.CONTROL.*"
        -- Requires LIST_COMPONENT access on the composite
        --  object implied by the partition.

procedure CLOSE_PARTITION(PH: in out PARTITION_HANDLE);

procedure GET_PARTITION_INFO(PH: in PARTITION_HANDLE;
    INFO: out PARTITION_INFO_BLOCK);
        -- Returns miscellaneous INFO about the partition,
        --  including the number of components currently in
        --  the partition, the FIRST, LAST, and NEXT component
        --  names (in ASCII lexicographic order), etc.

function GET_NEXT_COMPONENT(PH: in PARTITION_HANDLE) return STRING;
        -- This returns the name of the next component of the given
        --  partition, as a concatenated STRING of distinguishing
        --  attribute values.  The names
        --  are returned in ASCII lexicographic order.
```

Operations that create and delete components of a composite object implicitly modify its content. The name of the object specified to CREATE and CREATE_COMPOSITE determines the composite object in which it is created.

3.3.1.5.2 Processing

The content of a composite object is represented as a keyed access file, with the concatenation of the distinguishing attributes as the key. Because a multi-way B-tree keyed access file is used, the KAPSE provides fast (log N) access to components of even large composite objects.

As with simple extended objects (see above), composite object creation requires the creation of the extended object using the list access method, to hold the system-defined attributes such as COMPONENT_DA, ACCESS_CONTROL, CONTENT, WINDOW_XREF, etc.

3.3.1.5.3 Examples

```
CREATE_COMPOSITE("COMP", "MODULE  RELEASE_NUM");
CREATE(FH, "COMP.(MODULE=>DISPLAY, RELEASE_NUM=>1)", OUT_MODE);
CLOSE(FH);
OPEN(FH, "COMP.DISPLAY.1", IN_MODE); -- Using positional notation.
CLOSE(FH);
OPEN_PARTITION(PH, "COMP.*.1"); -- Scan through partition.
STR := GET_NEXT_COMPONENT(PH);
PUT_LINE("First component of COMP is: " & STR);
-- On the user's terminal should appear:
-- "First component of COMP is DISPLAY.1"
CLOSE_PARTITION(PH);
```

3.3.1.5.4 Special Requirements

Because this package is part of the KAPSE/Tool interface, it must faithfully enforce the KAPSE access control mechanisms. This is done by routines of the access control CPC (3.3.2.1) as part of pathname lookup, and extended object handle creation.

3.3.2 Access Control and Category (KAPSE.ACCECAT)

3.3.2.1 Window Objects

There are two kinds of windows: primary windows and secondary windows. Primary windows link an extended object to its enclosing composite object. Secondary windows allow an object to be viewed from a location other than the enclosing extended object. A thorough discussion of the tool interface to primary and secondary windows can be found in 3.2.4.3, subparagraphs 7 and 8.

3.3.2.1.1 Inputs and Outputs

A secondary window is created by the CREATE_WINDOW primitive, with parameters as follows:

a. WINDOW_PATH — Path where the window should be created

b. TARGET_PATH — Path to the target (relative to the creating program's context object).

c. PARTITION — Partition limitation, if any. This limitation is in addition to any already implicit in TARGET_PATH (i.e. TARGET_PATH'CURRENT_PARTITION).

d. TRANSLATION — Translation table, expressed as "(ext_role1,ext_role2,...) => (int_role1,int_role2,int_modifier,...),..." By default, WINDOW_PATH'CURRENT_ROLES are translated into TARGET_PATH'CURRENT_ROLES & TARGET_PATH'CURRENT_MODIFIERS. That is, the roles held by the creating program via the WINDOW_PATH are translated to the roles and modifiers held by the creating program via the TARGET_PATH. See Path-defined attributes in 3.2.4.3.4.

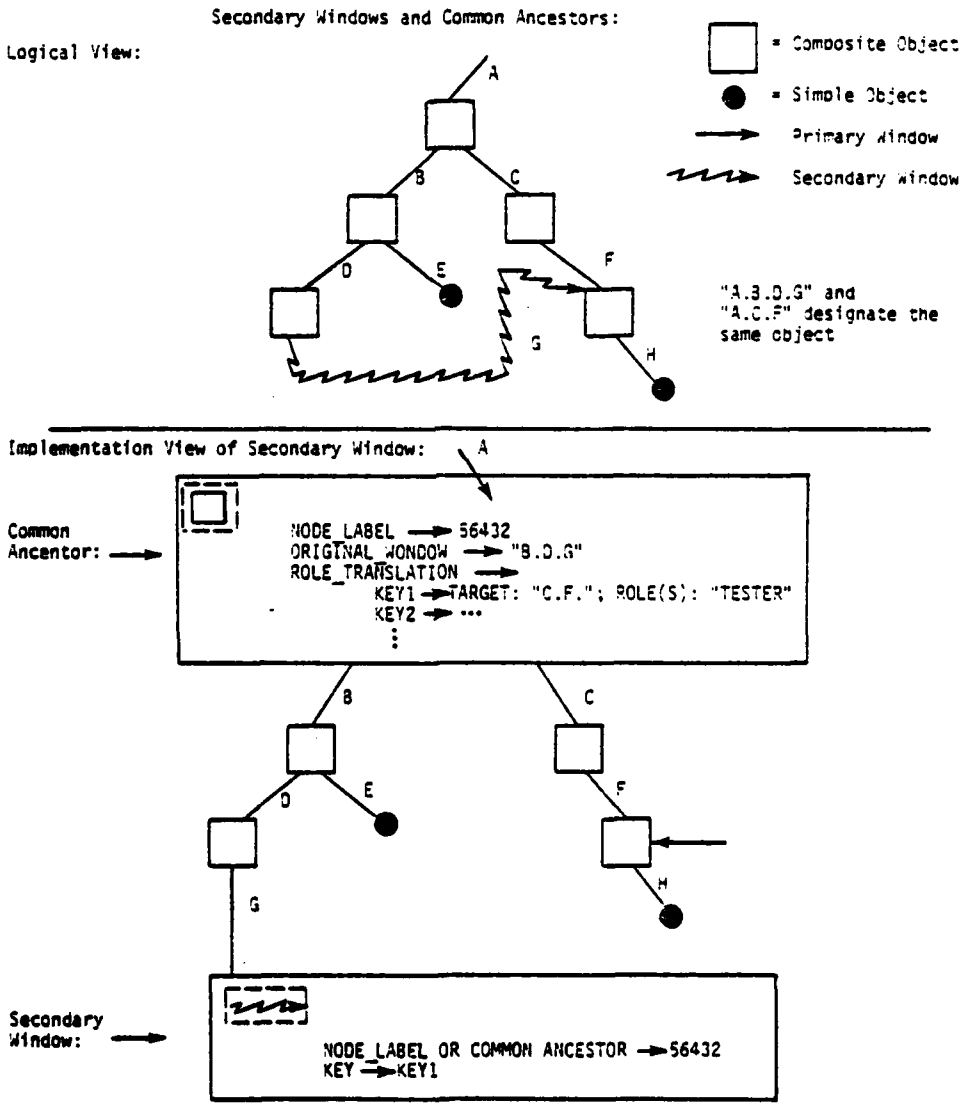
In any case, unless the window creator holds the _OWNER modifier, it is an error if this translation exceeds the roles or modifiers already held at the target.

e. COMMON_ANCESTOR_PATH — Path to the common ancestor (relative to the creating program's context object). It is an error if this is not an ancestor of the window, its target, and the common ancestors of all of the parents of the window (see 3.2.4.3.8.4). In this context, an extended object is considered an ancestor of itself.

This parameter will usually be defaulted. In that case, the extended object nearest to the target which satisfies the above constraints is chosen as the common ancestor.

See Figure 3-13 for an example of a secondary window.

Figure 3-13, Secondary Window with Its Common Ancestor:



The following user-visible routines are available for creating, deleting, querying, and revoking windows, as part of the KAPSE/Tool interface package WINDOW_OBJECTS:

B5-AIE(1).KAPSE(1)

```
Package WINDOW_OBJECTS is

type WINDOW_ID_STRING is new STRING;
        -- This type is used to identify windows,
        --  as a path to the common ancestor, and the
        --  window key there.
        -- The format of this string is irrelevant (though
        --  visible) to the user, because all relevant
        --  information is accessible via function calls.

type WINDOW_ATTRIBUTE_ENUM is
        (ORIGINAL_WINDOW, TARGET, PARTITION,
         ROLE_SET, MODIFIER_SET, PARENTS);

type WINDOW_FLAG_ENUM is
        (TRANSITORY, HAS_CHILDREN, REVOKED);
        -- The above two enumerations are used to
        --  request information given a WINDOW_ID_STRING
        --  (see 3.2.4.3.8.2).

procedure CREATE_WINDOW(WINDOW_PATH: in STRING;
                        TARGET_PATH: in STRING;
                        PARTITION: in STRING := "";
                        TRANSLATION: in STRING := "";
                        COMMON_ANCESTOR_PATH: in STRING := "");
        -- This creates a new secondary window, which
        --  may later be revoked by a parent window.

procedure DELETE_WINDOW(WINDOW_PATH: in STRING;
                        REVOKE_NOW: in BOOLEAN := FALSE);
        -- This deletes a window, rather than the
        --  object viewed through the window.
        -- It is possible to "revoke" the window
        --  at the same time, which invalidates
        --  any copies or descendents.

procedure COPY_WINDOW(OLD_WINDOW: in STRING;
                      NEW_WINDOW: in STRING);
        -- This makes a copy of an existing window.
        -- Copies cannot be individually revoked:
        --  when one goes, they all go.

procedure RENAME_WINDOW(OLD_PATH: in STRING;
                        NEW_PATH: in STRING);
        -- This is defined to be equivalent to
        --  COPY_WINDOW followed by
        --  DELETE_WINDOW(OLD_PATH, FALSE).

function  WINDOW_ID(WINDOW_PATH: in STRING;
                    HIGHEST_COMMON_ANCESTOR:
                        in STRING := "'ROOT")
                    return          WINDOW_ID_STRING;
        -- Return WINDOW_ID_STRING associated with
```

```
        -- window.  Routine will fail if cannot read
        -- common ancestor object, or if common ancestor
        -- above HIGHEST COMMON_ANCESTOR.

    function  NEXT_CHILD_WINDOW(PARENT_WINDOW: in STRING;
                                PREV_CHILD_ID: in WINDOW_ID_STRING;
                                HIGHEST_COMMON_ANCESTOR:
                                    in STRING := "^ROOT")
                                return           WINDOW_ID_STRING;
        -- This function iterates through the "children"
        --  of a window (see 3.2.4.3.8).
        -- In general, the children are those windows
        --  which the parent can legitimately revoke.
        -- The returned string is an identifier of
        --  the child.  The window id can be used to
        --  find the common ancestor, the path to
        --  the "ORIGINAL_WINDOW," the TARGET, etc.
        --  (3.2.4.3.8.2).

    function  WINDOW_ATTRIBUTE(WINDOW_ID: in WINDOW_ID_STRING;
                               ATTRIBUTE: in WINDOW_ATTRIBUTE_ENUM)
                               return        STRING;

    function  WINDOW_FLAG      (WINDOW_ID: in WINDOW_ID_STRING;
                               FLAG:      in WINDOW_FLAG_ENUM)
                               return        BOOLEAN;
        -- The above two functions return information
        --  associated with a window, identified
        --  by its WINDOW_ID_STRING.

    procedure REVOKE_WINDOW(PARENT_WINDOW:      in STRING;
                            WINDOW_ID:          in WINDOW_ID_STRING;
                            REVOKE_DESCENDENTS: in BOOLEAN := TRUE);
        -- Revoke window(s) designated by WINDOW_ID.
        -- If REVOKE_DESCENDENTS is TRUE, revoke all
        --  descendents also.
        -- Requires _OWNER or _OVERSEER modifier at parent.

    end WINDOW_OBJECTS;
```

This CPC also includes the KAPSE-internal routines to traverse windows as part of database pathnames, and to provide window cross-reference table maintenance during the copying and deleting of extended objects which include secondary windows.

The inputs to the window traversal routine are a handle on the window, and the current roles, modifiers, and partition limitations held outside the window. The outputs of the window traversal routine is a handle on the target, with an updated set of roles, modifiers, and partition limitations.

The inputs to the cross-reference maintenance routines are the handle on the object being manipulated, as well as the path to the new location if it is being copied. The outputs from the cross-reference maintenance routines are indications of whether the operation is permissible, and updated ANCESTOR_REFS attributes on the objects enclosing the object being deleted or created-by-copy.

3.3.2.1.2 Processing

The routines visible as part of the KAPSE/Host interface provide the basic creation and deletion operations on windows (see 3.2.4.3.8). These operations are implemented using the appropriate access methods to manipulate the list-access file, and the direct- and text-access components which are used to represent the information in primary and secondary windows.

These routines also manage the information recorded in the WINDOW_XREF attributes of extended objects used as common ancestors. All four access methods are used to manipulate the complex structure of this attribute. Generally, a new element must be added to the attribute at each window creation, and an element may be removed at window revocation. When a window is used as a parent, the appropriate HAS CHILDREN flag must be set.

The window routines also manage the ANCESTOR_REFS attribute present on each extended object with enclosed secondary windows which have common ancestors outside of the object. This attribute is a file keyed by the node labels of these referenced ancestors, with a count of the number of direct components which refer to each. As deletions and copies are made, the reference counts in these elements are adjusted. When a new label is added to an ANCESTOR_REFS set, or one is removed because its reference count goes to zero, a further adjustment must be made to the ANCESTOR_REFS attribute of the object enclosing this one. These adjustments may propagate all the way up to the labeled ancestor if this use of the common ancestor represents the first or last use of it in the whole database.

When a copy is to be made, these routines also check whether the new copy will remain a descendent of all of the ancestors mentioned in its ANCESTOR_REFS file. If this check fails, then the copy is not performed and an exception is raised in the user's program.

Note that enclosed secondary windows, with common ancestors also enclosed by an extended object, do not appear in the ANCESTOR_REFS attribute, and hence never interfere with the ability to copy such an extended object.

3.3.2.1.3 Examples

```
CREATE_WINDOW(".WORKSPACE", "SHUTTLE.NAVIGATION.INIT");
        -- This creates a convenient shorthand window
        --  named .WORKSPACE.

OPEN(FILE, ".WORKSPACE.SPEC");
        -- This is equivalent to:
        --  OPEN(FILE, "SHUTTLE.NAVIGATION.INIT.SPEC");

CREATE_WINDOW(".RESTRICTED_WORKSPACE", ".WORKSPACE",
    TRANSLATION => "*=>REVIEWER");
        -- Create sub-window limiting all to access rights
        --  given to the REVIEWER role.

OPEN(FILE2, ".RESTRICTED_WORKSPACE.SPEC");
        -- This may fail if SHUTTLE.NAVIGATION.INIT.SPEC
        --  doesn't give READ or ADD access to a REVIEWER.

CREATE_WINDOW(".SMALLER_VIEW", ".WORKSPACE.",
    PARTITION=>"(TEST_LEVEL=>2)");
        -- The window .SMALLER_VIEW only lets its user
        --  see objects with attribute TEST_LEVEL having
        --  a value of 2.
```

3.3.2.1.4 Special Requirements

Windows are used heavily in the AIE system to implement access control, history references, private objects, current view, etc. It is required that these routines which provide the fundamental as well as user-visible interfaces to windows work as efficiently as possible. Special mechanisms are provided to retain a "cache" record of recently traversed windows, so that actual walks up and down the database hierarchy can be minimized.

3.3.2.2 Category and User-defined Attributes

Attributes may in general be any kind of object. As such, the normal object manipulation routines will work on them. Nevertheless, separate packages have been defined to simplify access to certain kinds of attributes. In particular, special support is provided for numeric- and string-valued attributes and for the system-defined attributes (see below, 3.3.2.3, and 3.3.4.1).

3.3.2.2.1 Category Operations

3.3.2.2.1.1 Inputs and Outputs

The CATEGORY_DESCRIPTOR attribute is filled in by CREATE and CREATE_COMPOSITE, both of which can specify a category template as part of the optional FORM parameter. Complex category templates may be built up using normal object operations, or may be manipulated using one of the following routines designed to ease the process:

```
Package CATEGORY is

type CATEGORY_CLASS is (SIMPLE,COMPOSITE,CONTEXT,WINDOW,DEVICE);

procedure CREATE_CATEGORY_TEMPLATE(
    TEMPLATE:   in STRING;
    IDENTIFIER: in STRING;
    CLASS:      in CATEGORY_CLASS);
        -- Create a category template object,
        --  with the given CATEGORY identifier,
        --  for the given CLASS of extended object.

procedure DEFINE_VARIABLE_ATTRIBUTE(
    TEMPLATE:        in STRING;
    ATT_LABEL:       in STRING;
    ATT_INDEX:       in POSITIVE;
    ATT_CONSTRAINTS: in STRING := "");
        -- Define specified attribute to reside within
        --  extended object at specified index.
        --  Extended objects are represented by
        --  the list access method,
        --  and ATT_INDEX plus a system-defined offset
        --  will be the index into that list.
        -- The ATT CONSTRAINTS string is interpreted as
        --  described below.

procedure DEFINE_CONSTANT_ATTRIBUTE(
    TEMPLATE:          in STRING;
    ATT_LABEL:         in STRING;
    CONST_OBJECT_NAME: in STRING);
        -- Specify that attribute will be constant,
        --  and provide its value by naming object to
        --  be copied into slot in descriptor.

procedure DEFINE_CONSTANT_STRING_ATTRIBUTE(
    TEMPLATE:  in STRING;
    ATT_LABEL: in STRING;
    ATT_VALUE: in STRING);
        -- Specify that attribute will be constant,
        --  and provide its value as an ASCII string.
        -- This routine is just a convenience,
        --  and could be defined in terms of
        --  DEFINE CONSTANT ATTRIBUTE above.

function CONSTANT_ATTRIBUTE(
    TEMPLATE:  in STRING;
    ATT_LABEL: in STRING)
    return        BOOLEAN;
        -- Returns true if attribute is a constant.

end CATEGORY;
```

3.3.2.2.1.2 Processing

A category descriptor is created by copying a category template object. Both a descriptor, and a template for a descriptor have the same form, that of a keyed composite file, with the attribute label being the key. The above routines translate directly into operations on the category composite file, and its components.

When specifying an attribute constraint (ATT_CONSTRAINT above), the limitation may be to a list of values (e.g. "source object executable") or to a range of values (e.g. "0 .. 10" or "1 .. *") with "*" meaning plus or minus infinity, as appropriate. "* .. *" restricts the value to be numeric. An attempt to violate the constraint on an attribute is automatically caught by the KAPSE, and aborted.

As part of the delivered KAPSE, category templates will be provided for such common composite objects as an Ada library, a user mailbox, and a typical user top-level directory.

3.3.2.2.1.3 Special Requirements

3.3.2.2.2 Operations on Numeric and String-valued Attributes

3.3.2.2.2.1 Inputs and Outputs

Attribute values which are a simple ASCII string are actually represented internally as a text file. This allows the strings to grow arbitrarily long, but makes certain simple operations clumsy. To alleviate this problem, a separate package is defined to ease access to such string-valued attributes:

```
package STRING_ATTRIBUTES is

procedure SET_ATTRIBUTE(NAME: in STRING; ATT_LABEL: in STRING;
    ATT_VALUE: in STRING);
        -- By setting an attribute value to the null STRING,
        --  the attribute is effectively deleted.
        -- Requires add/delete access.
```

```
function GET_ATTRIBUTE(NAME: in STRING; ATT_LABEL: in STRING)
    return STRING;
        -- Attribute value returned as null STRING if not
        --  previously SET.
        -- Numeric-valued attributes returned as their
        --  decimal representation.
        -- Requires read access.

function GET_ALL_ATTRIBUTES(NAME: in STRING) return STRING;
        -- Return all non-null, string- or numeric-valued
        --  attributes, in a label=>value list.

end STRING_ATTRIBUTES;
```

Furthermore, certain attributes are limited to numeric values. These include certain category-defined attributes, and system-defined attributes of primitive files. Also, user-defined attributes, though generally represented as a string, may frequently be more easily manipulated as a number. The following package is provided for the manipulation of such numeric-value attributes:

```
Package NUMERIC_ATTRIBUTES is

procedure SET_ATTRIBUTE(NAME: in STRING; ATT_LABEL: in STRING;
    ATT VALUE: in INTEGER);
        -- Set the designated attribute to the
        --  specified numeric value.
        -- If the attribute label is user-defined,
        --  still use a string to represent its
        --  numeric value.
        -- Requires add/delete access.

function GET ATTRIBUTE(NAME: in STRING; ATT_LABEL: in STRING)
    return INTEGER;
        -- Return the current value of a numeric-valued
        --  attribute.
        -- Exception if attribute does not have a numeric value.
        -- Requires read access.

end NUMERIC_ATTRIBUTES;
```

3.3.2.2.2.2 Processing

String and numeric attributes are actually stored as text- or direct-access files. When SET_ATTRIBUTE or GET_ATTRIBUTE is called, the full pathname of the attribute is created by concatenating NAME and ATT LABEL separated by a "tic" (apostrophe), and then the file is located and manipulated with the appropriate access method (see 3.3.1.3).

In the case of SET_ATTRIBUTE, the contents of the ATT_VALUE string or number are simply written out to the file, and then the file is released. For GET_ATTRIBUTE on a string, a string of length CHAR_COUNT(text_file) is declared, the characters are read from the file, the file is released, and the string returned. For GET_ATTRIBUTE of a number, the attribute is read as a string if it is text, and then converted using INTEGER'VALUE [LRM].

Given their label as a string, keyed access files are used to locate the attribute descriptors or values. Hence, SET_ATTRIBUTE and GET ATTRIBUTE operate rapidly (log N) even when the number of attributes is large.

GET_ALL_ATTRIBUTES scans the object's category descriptor and USER_DEFINED_ATTRIBUTES attribute looking for string-valued attributes, and concatenates strings in the label=>value form together.

3.3.2.2.2.3 Examples

```
SET_ATTRIBUTE("TEST_FILE", "PURPOSE", "FUN");
SET_ATTRIBUTE("TEST_FILE", "CHECK_LEVEL", "1");
SET_ATTRIBUTE("XYZ", "PURPOSE", "FUN");
declare
        S: constant STRING := GET_ATTRIBUTE("XYZ", "CHECK_LEVEL");
                -- S is now the null STRING.
        AA: constant STRING := GET_ALL_ATTRIBUTES("TEST_FILE");
begin
        PUT_LINE(AA);
        -- Output will be: "PURPOSE=>FUN,CHECK_LEVEL=>1"
end;
```

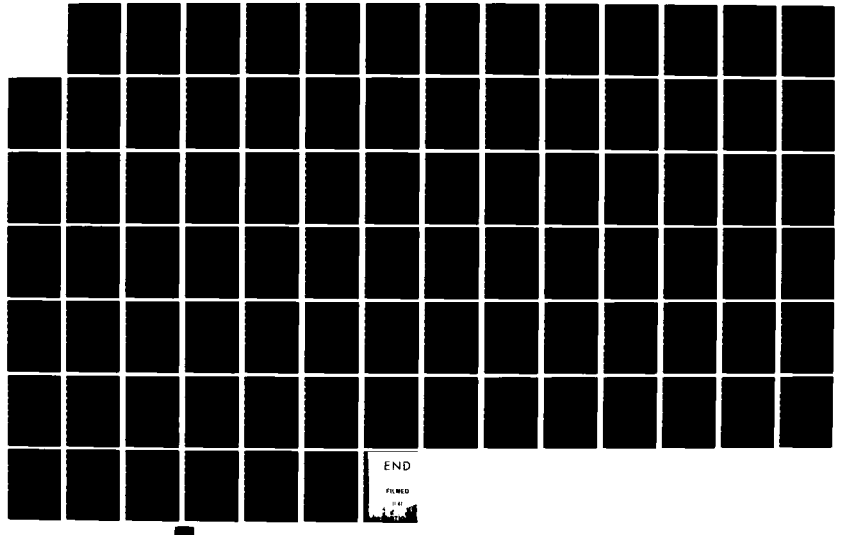
3.3.2.2.2.4 Special Requirements

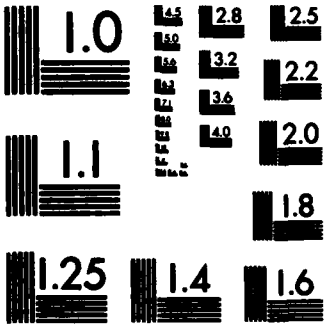
3.3.2.3 Access Control

3.3.2.3.1 Static Access Control

AD-A134 092 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR ADA INTEGRATED ENVIRONMENT..(U) INTERMETRICS INC CAMBRIDGE MA 12 NOV 82 IR-678-2 F30602-80-C-0291 2/2

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

3.3.2.3.1.1 Inputs and Outputs

As part of the KAPSE/tool interface, the following primitives are available for manipulating the extended object attributes relevant to access control, including ACCESS CONTROL and ROLES, and implicitly the primary or secondary window used to reach the extended object:

```
Package ACCESS_CONTROL is

subtype ROLE_STRING is STRING;
subtype ACCESS_RTS_STRING is STRING;

procedure SET_ROLE_ACCESS(OBJNAME:     in STRING;
                          ROLE:        in ROLE_STRING;
                          ACCESS_RTS: in ACCESS_RTS_STRING);
        -- Set list of access rights associated with given
        --  role.  Format for ACCESS_RTS is
        --  comma-separated list of access right
        --  names, with operations and channels specified
        --  using an embedded list.
        -- Requires _OWNER modifier, or _OVERSEER modifier
        --  with READ access.

function ROLE_ACCESS(OBJNAME: in STRING;
                     ROLE:    in ROLE_STRING)
                     return       ACCESS_RTS_STRING;
        -- Return list of access rights associated with
        --  specified role for the designated object.
        -- Returned STRING is comma-separated
        --  list of access right names and embedded
        --  list of operations/channels.
        -- Requires READ access to the object.

function ALL_ROLES(OBJNAME: in STRING) return STRING;
        -- Return list of roles with any access rights
        --  explicitly defined for this object.
        -- Requires READ access to the object.

procedure CREATE_ROLE(NEW_ROLE: in ROLE_STRING;
                      WHERE:    in STRING);
procedure DELETE_ROLE(NEW_ROLE: in ROLE_STRING;
                      WHERE:    in STRING);
        -- Edit the ROLES attribute.
        -- Must have _OWNER or read-able _OVERSEER.
        -- WHERE must "end" on an extended
        --   object, and the last step must be
        --   via its primary window.
        -- Implicit ADOPT_ROLE/ABANDON_ROLE is performed.
```

```
   procedure ADOPT_ROLE   (ROLE:     in ROLE_STRING;
                           WHERE:    in STRING := "'CURRENT_DATA");
   procedure ABANDON_ROLE(ROLE:      in ROLE_STRING;
                           WHERE:    in STRING := "'CURRENT_DATA");
           -- ADOPT requires "add" right at window
           --  and _OWNER inside target;
           --  ABANDON requires "delete" at window.

   procedure GIVE_ROLE    (ROLE:     in ROLE_STRING;
                           TO_ROLE: in ROLE_STRING;
                           WHERE:    in STRING := "'CURRENT_DATA");
           -- Requires "add" at window, and giver
           --  must hold ROLE (or _OWNER
           --  modifier) at target of window.
           -- TO_ROLE is a role outside the window
           --  receiving the additional
           --  ROLE inside the target.

   procedure SET_MODIFIER(MODIFIER: in MODIFIER_STRING;
                           WHERE:    in STRING := "'CURRENT_DATA";
                           TO:       in BOOLEAN := TRUE);
           -- Only legal to set _READ_ONLY true,
           --  or to set _OWNER or _OVERSEER false,
           --  unless already have _OWNER modifier.

   end ACCESS_CONTROL;
```

In addition to this user-visible package, the access control CPC provides routines for use within the KAPSE, to lookup pathnames, and verify access rights for primitive operations. The inputs for these routines are generally the pathname, a handle on the current program context object (where the pathname implicitly starts), and an identification of what access rights are about to be exercised.

The outputs from these KAPSE-internal routines are typically a handle on the extended object and the particular file selected by the pathname, or a failure indication indicating that an access control violation has occurred.

3.3.2.3.1.2 Processing

The access control attribute is represented by a table indexed by role index, with each element identifying the associated access-rights and operations/channels. The primitive access method routines (see 3.3.1.3) are used to perform the appropriate manipulations, after verifying that the running program has the appropriate access role modifiers/rights.

When the CPC wishes to check the legality of an operation, it consults the access control attribute of the object, with a set of roles and the number of the required access right. The

required access right must be granted to at least one of the roles for the check to succeed.

The lookup function of this CPC does pathname interpretation (see 3.2.4.3.3). It implements the rules for content- and target-defined attributes (see 3.2.4.3.4) as part of pathname interpretation. It uses routines of the window object CPC to locate the target of a window, and translate the roles as appropriate.

3.3.2.3.1.3 Examples

```
SET_ROLE_ACCESS("ALPHA", ROLE=>"WORLD",
    ACCESS_RTS=>"READ,ADD");
        -- Give all users with WORLD window over simple
        --  object ALPHA, access rights READ and ADD.

SET_ROLE_ACCESS("BETA", ROLE=>"PROJECT",
    ACCESS_RTS=>"READ,OPERATE=>(LIST,EXTRACT)")
        -- Give all users with PROJECT role inside
        --  object BETA, right to READ, and invoke
        --  the LIST and EXTRACT operations.

TEXT_IO.PUT( GET_ROLES("BETA").);
        -- Will print "PROJECT" if above SET_ROLE_ACCESS
        --  is the only one in effect for BETA.
```

3.3.2.3.1.4 Special Requirements

This CPC in conjunction with the WINDOW_OBJECT CPC forms the heart of the KAPSE access protection mechanism. It is required that this CPC be thoroughly tested to ensure that no possibilities for protection violation remain within the access control implementation.

3.3.2.3.2 Dynamic Access Synchronization

3.3.2.3.2.1 Inputs and Outputs

The following primitives are used to effect synchronization among multiple Ada programs attempting to access overlapping parts of the database:

```
Package ACCESS_SYNCHRONIZATION is

type OBJECT_HANDLE is limited private;
        -- Reserved objects are referred to by
        --  object handles, created within the
        --  context object of the reserving
        --  program.

type RESERVE_MODE is (WRITE_ORIGINAL, WRITE_COPY, READ_ORIGINAL,
    READ_COPY, SHARED_STREAM, SHARED_RANDOM);
        -- WRITE_ORIGINAL prevents all access except
        --  READ/WRITE_COPY.
        -- READ_ORIGINAL prevents all write access.
        -- READ/WRITE_COPY never interferes, but may also be
        --  reading/writing soon-to-be-obsolete data.
        -- SHARED_STREAM causes WRITE_ORIGINAL reservation
        --  only at the time of actual READ or WRITE.
        --  Stream READ always reads the first defined element of
        --  the object, and then advances FIRST to the next element,
        --  and requires READ/WRITE  or CONSUME access.
        --  Stream WRITE always appends a new element at the end of
        --  the object and advances LAST.
        -- SHARED_RANDOM causes a reserve (WRITE_ORIGINAL
        --  or READ_ORIGINAL) only at the time of actual READ or WRIT

procedure RESERVE(HANDLE: in out OBJECT_HANDLE;
                  NAME:        in STRING;
                  MODE:        in RESERVE_MODE;
                  TIME_LIMIT: in DURATION := DURATION'LAST);
        -- The object named is reserved
        --  according to the given RESERVE_MODE.
        -- If the RESERVE is not immediately possible
        --  due to a conflicting RESERVE, the caller is delayed
        --  up to the specified TIME_LIMIT, when a TIME_OUT
        --  exception will occur.

procedure RELEASE(HANDLE: in out OBJECT_HANDLE);
        -- RELEASE after RESERVE for WRITE_ORIGINAL causes
        --  modifications made since the RESERVE to become
        --  permanent.
        -- RELEASE after READ_ORIGINAL allows waiting writers to
        --  proceed to RESERVE.
        -- RELEASE after READ/WRITE_COPY throws away the logical
        --  COPY made for the purpose of private work.

procedure ABORT_RESERVE(HANDLE: in out OBJECT_HANDLE);
        -- ABORT_RESERVE after WRITE_ORIGINAL returns the
        --  reserved object or partition to its original
        --  pre-RESERVE state.
        -- For all other modes, ABORT_RESERVE is
        --  equivalent to RELEASE.
```

```
function HANDLE_NAME(HANDLE: in OBJECT_HANDLE)
                        return      STRING;
        -- This function returns a pathname string to
        --  be used for opening the reserved object,
        --  or passing the reserved handle off
        --  to a subsidiary program.
        -- The pathname is always of the form
        --  "^OPEN_HANDLES.xx" where xx is an
        --  index into the OPEN_HANDLES list.

end ACCESS_SYNCHRONIZATION;
```

Besides these explicit synchronization calls, CREATE of a simple object, OPEN of a simple object, and OPEN_PARTITION result in implicit reserves. By default, OPEN for input only and OPEN_PARTITION do a READ_COPY reserve. CREATE and OPEN for output do an WRITE_ORIGINAL reserve. The default reserve may be overridden by additional information in the FORM STRING passed to OPEN, providing for READ_ORIGINAL reserve instead of READ_COPY, or selecting SHARED_STREAM or SHARED_RANDOM, in which case, an automatic RESERVE/RELEASE takes place around each READ and WRITE operation to the object.

3.3.2.3.2.2 Processing

After an Ada program performs a RESERVE, it may perform a sequence of operations using the reserved handle without interference from other programs. When the sequence is complete, the program may RELEASE or ABORT_RESERVE. Each RESERVE starts by making a logical COPY of the reserved object. Modifications and accesses performed between RESERVE and RELEASE use this logical COPY, preserving the integrity of the original object.

The CPC implements RESERVE/RELEASE at a low level to allow efficient detection of conflicting reservations. When READ/WRITE_ORIGINAL reservation of all or part of an object is requested, this CPC determines whether a conflicting reserve is already in progress. If so, the new reserve is delayed up to the TIME_LIMIT. If not, this CPC records the reservation, and for WRITE_ORIGINAL, creates a logical copy where the actual changes will be made. READ/WRITE_COPYers never need to check for conflicting reserve. They simply make a logical copy for their own use of whatever is available, which may be somewhat out of date.

3.3.2.3.2.3 Special Requirements

Access synchronization routines are called implicitly by every open/create operation on simple objects. It is required that the creation of the reserved handles, including any logical

copies, be as efficient as possible to preserve the overall performance of the KAPSE.

3.3.3 Multiple Program Management (KAPSE.MULTPROG)

3.3.3.1 Program Loading

3.3.3.1.1 Inputs and Outputs

The following package is provided to interface to the host loading, initiation, memory-allocation, and time-sharing facilities, as part of the KAPSE/Host interface:

```
package PROGRAM_LOADING is

procedure LOAD_PROGRAM(LOAD_MODULE_NAME: in STRING;
    ID: out PROGRAM_ID);
        -- This procedure loads and initiates the designated
        --  program.  The returned PROGRAM_ID may
        --  be used later to communicate with the
        --  program.
        -- The LOAD_MODULE_NAME is a database pathname
        --  that identifies a simple object suitable for
        --  loading by the host system.  The history
        --  attribute of the load module uniquely identifies
        --  the state of its content, and the implementation
        --  may attempt to share code for multiple executions
        --  using the same load module.  Extra effort will
        --  be made to share the code of frequently
        --  executed programs.

procedure UNLOAD_PROGRAM(ID: in PROGRAM_ID);
        -- This procedure frees any space allocated to the
        --  program identified by ID, and performs any
        --  clean-up that may be required on the host
        --  system to eradicate the program.

procedure GET_STORAGE (AMOUNT: in STORAGE_AMOUNT;
    STORAGE: out STORAGE_PTR);
        -- This procedure allocates storage of the amount
        --  specified by the caller, and returns an access
        --  value that identifies the storage that was
        --  allocated.
        -- The allocated storage contains inside it a header
        --  that indicates the size that was allocated.

procedure FREE_STORAGE (STORAGE: in STORAGE_PTR);
        -- This procedure returns storage to the system.
        --  The size of the returned storage is
        --  determined from a field in the header of the
        --  storage area.
```

```
end PROGRAM_LOADING;
```

3.3.3.1.2 Processing

This CPC has three distinct subdivisions.

a. GET_STORAGE and FREE_STORAGE, which are called by the run-time system (see 3.3.5.2) in response to requests made by the program.

b. LOAD_PROGRAM and UNLOAD_PROGRAM, which are called by Program Invocation routines of KAPSE.MULTPROG (see 3.3.3.3), and are used to start a program running and to clean-up any residue left behind by a completed Ada program.

c. This CPC also guarantees that a program that has been loaded via LOAD_PROGRAM will be allocated sufficient processing, memory, and other resources to complete. There is no procedure call associated with this requirement; rather, the scheduling happens automatically.

3.3.3.1.3 Processing for VM/SP

This CPC allocates memory within the virtual machine, sets up the segmentation and page maps appropriately, and then loads the data from the load module into memory. When sharing is warranted, the pure portion of the load module is separately located, and multiple invocations of the same program will simply map it in.

3.3.3.1.4 Processing for PE OS/32

Unshared Ada programs are initiated by loading a pre-initialized OS/32 task image whose sharable pure segment includes the standard Ada run time system. The start-up code of the task reads the blocks of code and data into its impure segments.

A limited number of host files are created and allocated when the KAPSE is installed, for the purpose of holding OS/32 task images with sharable segments. When sharing is warranted, the load module is copied into a file in Task Establisher Task (TET) format; this file is then used for task loading. These files are re-used dynamically on a "Least Recently Used" basis.

3.3.3.1.5 Special Requirements

3.3.3.2 Low Level KAPSE/Program Communication

3.3.3.2.1 Inputs and Outputs

The following package is part of the KAPSE/Host interface, and provides the basic mechanism for communication between the KAPSE and the user programs, in a host-independent manner:

B5-AIE(1).KAPSE(1)

```
package KAPSE_PROGRAM_COMMUNICATION is

-------------------------------------

type PROGRAM_ID is private;
        -- The program id for each program is unique,
        --  and is assigned by package PROGRAM_LOADING
        --  (see above).

type REQUEST_INDEX is
    INTEGER range 1..<<Implementation Dependent>>;
        -- Each Ada program is limited to a specific number
        --  of outstanding KAPSE_CALLS (the presence of
        --  multi-tasking implies that there may be several
        --  KAPSE_CALL's outstanding at once).  Each
        --  KAPSE_CALL is associated with an integer that
        --  identifies which of the permitted requests
        --  was responsible for the call.

type KAPSE_PACKAGE_ENUM is
    (<<List of package id's for all
       packages exported by KAPSE>>);
        -- Each KAPSE interface package is associated with
        --  a unique enumeration.  The enumeration is used
        --  as a discriminant to the MESSAGE_RECORD and
        --  RESULTS_RECORD types below.

type MESSAGE_RECORD(KIND: KAPSE_PACKAGE_ENUM) is <<TBD>>;
        -- This type defines the structure of messages
        --  passed via the KAPSE PROGRAM_COMMUNICATION
        --  routines.  The structure varies according
        --  to the kind of kapse call.

type RESULTS_RECORD(KIND: KAPSE PACKAGE_ENUM) is <<TBD>>;
        -- This type defines the structure of results
        --  passed via the KAPSE_PROGRAM_COMMUNICATION
        --  routines.  The structure varies according
        --  to the kind of kapse call.

-------------------------------------

package USER_VERSION is

procedure KAPSE CALL (MESSAGE: in MESSAGE_RECORD;
    RESULTS: out RESULTS_RECORD);
        -- This procedure signals to the KAPSE via an
        --  interrupt that a message should be sent
        --  across the KAPSE protection boundary.
        --  The procedure waits for the KAPSE to send
        --  results back across the protection boundary.
        -- The caller is suspended until the results
        --  have been passed back.
```

```
end USER_VERSION;

-------------------------------------

package KAPSE_VERSION is

procedure RECEIVE_REQUEST (ID: out PROGRAM_ID;
    INDEX: out REQUEST INDEX;
    MESSAGE: out MESSAGE RECORD);

procedure RETURN_RESULTS (ID: in PROGRAM_ID;
    INDEX: in REQUEST INDEX;
    RESULTS: in RESULTS RECORD);

end KAPSE_VERSION;

-------------------------------------

end KAPSE_PROGRAM_COMMUNICATION;
```

3.3.3.2.2 Processing

This CPC implements a user "communication task" and a KAPSE "communication task" to actually send messages across the KAPSE protection boundary. These tasks are also part of the KAPSE/Host interface implementation and are implemented differently on different hosts. A table outlining the processing done on each side of the protection boundary follows:

KAPSE/Program Communicaton Mechanism

Program :

* KAPSE_CALL:

  1) KAPSE_CALL makes an entry call on the user communication task to send the message and the results address to the KAPSE. The request index is returned.

  2) The user communication task determines the program id of the caller, finds an available request index, and sends the program id, the request index, the message and the results address to the KAPSE. It returns the allocated request index to the caller.

  3) The KAPSE_CALL makes an entry call on the the member indicated by the request index of the GetResults entry of the user communication task and is normally blocked.

  4) A message interrupt causes the user communication task to do an accept on the entry member associated with the request index for the completed KAPSE_CALL.

  5) Awoken by the end of rendezvous, the KAPSE_CALL returns to the caller.

KAPSE:

* RECEIVE_REQUEST

  1) A "server" task calls RECEIVE_REQUEST which does an entry call on the KAPSE communication task's "GetRequest" entry and is normally blocked.

  2) A message interrupt causes a call to be made on the KAPSE communication task which copies the message across the KAPSE protection boundary into the KAPSE.

  3) The KAPSE communication task then does an accept on its GetRequest entry and provides the

caller with the program id, the request index, the message, and the result address of the associated message.

4) The server, awoken by the accept, services the request (as specified in the message).

* RETURN_RESULTS

5) Server calls RETURN_RESULTS which does an entry call on the KAPSE communication task's "SendResults" entry which copies the results back into the user's space and causes a message interrupt (tagged with the request index) to be sent to the appropriate program.

3.3.3.2.3 Processing for VM/SP

The KAPSE/Host interface under VM/SP implements this KAPSE/user program communication using the SVC instruction. The KAPSE/Host interface has direct access to the address space of the user program, so the data may be copied across using the MVC instruction.

3.3.3.2.4 Processing for PE OS/32

Communication between the KAPSE OS/32 task and user program OS/32 tasks use the OS/32 task message facility. Pseudo interrupts are provided to the receiving task when a message is ready.

For large transfers, OS/32 provides the ability to send and receive open file handles. If the overhead of messages becomes unwieldy in a running MAPSE, it will be possible to switch to a method of data transfer involving writing to a scratch file from one task, and then reading the data back in the receiving task.

3.3.3.2.5 Special Requirements

The KAPSE_PROGRAM_COMMUNICATION package is used for all communication between the KAPSE and user programs, and is critical to the efficiency of KAPSE system calls in general.

3.3.3.3 Program Invocation and Control

3.3.3.3.1 Program Context

3.3.3.3.1.1 Inputs and Outputs

Each activation of a program has associated with it exactly one program context object. The following primitives are available to create new activations of a program with its new context object, as well as suspend and resume the running program. The context object is initialized from parameters, windows, and other attributes inherited from the invoker, and with a window back on the executable program object.

```
Package PROGRAM_INVOCATION is

subtype PARAMS_STRING  is STRING;
subtype RESULTS_STRING is STRING;

function CALL_PROGRAM(PROGRAM_PATH: in STRING;
                      PARAMETERS:   in PARAMS_STRING;
                      CONTEXT_NAME: in STRING := ".SUB_CONTEXT";
                      STD_IN:       in TEXT_IO.FILE_TYPE :=
                          CURRENT_INPUT;
                      STD_OUT:      in TEXT_IO.FILE_TYPE :=
                          CURRENT_OUTPUT)
                      return            RESULTS_STRING;
        -- This function invokes an executable program
        --  context or command language script as
        --  though it were a sub-program of
        --  the calling program.
        -- PROGRAM_PATH is the access path to the program/script.
        -- PARAMETERS is a comma-separated
        --  list of parameters for the
        --  program, using positional or keyword
        --  notation (eg., "A,B,EXTRA=>C" ).
        -- The optional parameter CONTEXT_NAME specifies
        --  the LOCAL_NAME for the context object
        --  created for the called program.
        -- The returned RESULTS_STRING is a
        --  comma-separated list of the out parameter
        --  values of the called program.
        -- If the called program is actually a function,
        --  the result is returned as though it were
        --  an out parameter labeled RETURN
        --  (eg., "RETURN=>1423" ).
        -- By default, the current text input and output for
        --  the calling program become the standard
        --  text input and output for the called program.
        -- All attributes of the caller's context with
        --  INHERIT flag set are copied
        --  into the sub-context created.

function PROGRAM_SEARCH(PROG_NAME: in STRING) return STRING;
        -- This function looks for an executable program
        --  context or command language script with
        --  name PROG_NAME in each of the composite
        --  objects specified in the caller's PROGRAM_SEARCH_LIST.
        -- The returned STRING is the full access path to
        --  the program context of script, ready to
        --  be passed to CALL_PROGRAM above.
        -- The PROGRAM_SEARCH_LIST is an attribute of
        --  the caller's context object.  It is set using
        --  SET ATTRIBUTE and specified as a
        --  comma-separated list of composite object names.
```

```
   procedure INITIATE_PROGRAM(PROGRAM_PATH: in STRING;
                              PARAMETERS:   in PARAMS_STRING;
                              CONTEXT_NAME: in STRING;
                              STD_IN:       in TEXT_IO.FILE_TYPE;
                              STD_OUT:      in TEXT_IO.FILE_TYPE);
           -- This procedure invokes a program or
           --  script exactly like CALL_PROGRAM,
           --  except that the caller is not suspended
           --  until completion, and no defaults are
           --  provided for CONTEXT_NAME, STD_IN, or
           --  STD_OUT.

   function AWAIT_PROGRAM(CONTEXT_NAME: in STRING;
                          TIME_LIMIT:   in DURATION :=
                              DURATION'LAST)
                          return           RESULTS_STRING;
           -- This function waits for the completion
           --  of the specified program context object,
           --  up to the specified TIME_LIMIT.
           -- The returned STRING is as in CALL_PROGRAM.

   procedure EXIT_PROGRAM(RESULTS:              in RESULTS_STRING;
                          ABORT_SUB_CONTEXTS: in BOOLEAN := FALSE);
           -- This procedure exits a program, either
           --  waits for its sub-contexts or aborts them,
           --  and then returns the results to the invoker.

   procedure SUSPEND_PROGRAM(CONTEXT_NAME: in STRING);
           -- The program executing in the named context is stopped,
           --  allowing the state of the execution to be examined,
           --  or a debugger to be initiated to control or trace
           --  further execution of the program.
           -- Normal tasks of the program are made dormant, but
           --  the run-time system continues to respond to inter-
           --  program communication on channels zero and one.

   procedure RESUME_PROGRAM(CONTEXT_NAME: in STRING);
           -- The program associated with the named context is
           --  restarted. The program must have been previously
           --  initiated and then suspended.
```

(continued below)

3.3.3.3.1.2 Processing

A program context is a composite object using a single component distinguishing attribute LOCAL_NAME and with certain standard windows and objects as components. In particular, every context includes a window attribute labeled CURRENT_DATA, which provides the main link to the permanent part of the database. The CURRENT_DATA window may be shifted to view other parts of the

database using the CHANGE_VIEW primitive (see 3.3.3.6 below). A running program has an implicit OWNER window on its program context.

The program integration facility [AIE(1).PIF(1)] creates executable program objects and by default deposits them in their associated Ada program library. If the program is to be used by many users, it will be copied to a central repository of executable programs (eg., TOOLS component of the root). When an executable program is called or initiated, this CPC creates a new context object with a window attribute labeled PROGRAM whose target is the executable program object, allowing the running program to refer to attributes of the program object via a pathname like "'PROGRAM'HELP_FILE."

When a command language script is called, the KAPSE invokes the command language processor identified by the PROCESSOR attribute of the program object, and passes the name of the object containing the script as an additional parameter.

3.3.3.3.1.3 Special Requirements

Program invocation is used heavily within the AIE, because of the basic toolkit approach. Sophisticated tools can be built up out of simpler fragments using program invocation as the primary composition technique. It is required that the implementation of program invocation be as time-efficient as possible to preserve overall performance of the AIE.

3.3.3.3.2 Parameter Passing

3.3.3.3.2.1 Inputs and Outputs

Parameters are passed to a program context by CALL_PROGRAM and INITIATE_PROGRAM (see above) as a comma-separated list using positional or keyword notation. For example:

```
CALL_PROGRAM("COMPILE", "QSORT,MYLIB,OPTIM=>TIME");
```

Internally, these parameters are passed as the value of an attribute of the created program context, labeled PARAMETERS. This attribute is then retrieved by the called program's preamble [AIE(1).PIF(1)], by GET_ATTRIBUTE(".","PARAMETERS").

At the end of execution, values of out parameters are rewritten by the called program to the RESULTS attribute using SET_ATTRIBUTE, and are returned to the caller as the results string of CALL_PROGRAM or AWAIT_PROGRAM. If the called program

is a function, the returned string is of the form "RETURN=>return_value." If the program ends due to an unhandled exception, the returned string will be "EXCEPTION=>exception_id."

The following function is defined to facilitate extracting a single parameter from the string returned by GET_ATTRIBUTE, CALL_PROGRAM, or AWAIT_PROGRAM:

```
function PICK_PARAM(PARAMETERS: in STRING; PARAM_NAME: in STRING;
    POSITION: in INTEGER := 0; DEFAULT: in STRING := "")
    return STRING;
        -- This function extracts the specified parameter from
        --  the given parameter string, as might be returned
        --  by GET_ATTRIBUTE(".", "PARAMETERS").
        -- PARAM_NAME may be null or POSITION may be zero,
        --  but not both.  The DEFAULT string is returned if
        --  no parameter is present in PARAMETERS at the
        --  designated POSITION or labeled by the
        --  specified PARAM_NAME.
```

3.3.3.3.2.2 Processing

The list of parameters is represented as the attribute PARAMETERS of the program context object. The function PICK_PARAM is provided to parse the parameter list, and does so by simply scanning through the PARAMETERS string supplied, looking for "PARAM_NAME =>" if PARAM_NAME is not null, or the unlabeled argument number POSITION. If neither is present, the supplied DEFAULT string is returned.

3.3.3.3.2.3 Special Requirements

3.3.3.3.3 Private Object Operations

3.3.3.3.3.1 Inputs and Outputs

The following primitives are available for creating and invoking private object operations:

```
function INVOKE_OPERATION(PRIV_OBJ:      in STRING;
                          OPERATION:     in STRING;
                          PARAMETERS:    in PARAMS STRING;
                          CONTEXT_NAME:  in STRING := ".SUB_CONTEXT";
                          STD_IN:        in TEXT_IO.FILE_TYPE :=
                              CURRENT INPUT;
                          STD_OUT:       in TEXT_IO.FILE_TYPE :=
                              CURRENT_OUTPUT)
                          return             RESULTS_STRING;
        -- This routine attempts to invoke the specified
        --  operation.
        -- The operation will fail if the PRIV_OBJ does
        --  not have an OPERATIONS attribute,
        --  or the caller does not have access to it.
        -- The returned RESULTS_STRING is the
        --  out parameters or the return value
        --  of the operation.

end PROGRAM_INVOCATION;
```

3.3.3.3.3.2 Processing

Private objects are simply objects with an OPERATIONS attribute, which is a window on a composite object full of operations (i.e. executable program objects). When the user calls INVOKE_OPERATION, the KAPSE constructs the pathname for the operation context object as PRIV_OBJ & "^OPERATIONS." & OPERATION. It then creates a context object, gives it a window on PRIV_OBJ called "^IMPLICIT_OBJECT" of role "OWNER," and prepends "^IMPLICIT_OBJECT," to the parameter list (e.g. if PARAMETERS is "A,B" then it passes "^IMPLICIT_OBJECT,A,B" as the full parameter list to the operation).

3.3.3.3.3.3 Special Requirements

3.3.3.3.4 Interprogram Communication

3.3.3.3.4.1 Inputs and Outputs

Interprogram communication is performed by special operations on the associated program context objects. The routines of this package provide the equivalent of an inter-program rendezvous, receiving the parameters, and returning results, in analogy with task entry calls, just as CALL_PROGRAM provides an analogy to subprogram calls.

B5-AIE(1).KAPSE(1)

This package is part of the KAPSE/Tool interface:

```
with PROGRAM_INVOCATION; use PROGRAM_INVOCATION;
Package INTER_PROGRAM_COMMUNICATION is

function IPC_ACCEPT(CHANNEL_NAME: in STRING;
                    TIME_LIMIT:   in DURATION :=
                        DURATION'LAST)
                    return          PARAMS_STRING;
        -- This routine accepts the next waiting
        --  IPC_ENTRY_CALL for this channel.
        --  If none are already waiting, this
        --  will suspend up to the TIME LIMIT.

procedure IPC_END_RENDEZVOUS(CHANNEL_NAME: in STRING;
                             RESULTS:      in RESULTS_STRING);
        -- This routine is called after an IPC_ACCEPT,
        --  to allow the IPC_ENTRY_CALL to proceed,
        --  with the RESULTS provided.

function  IPC_ENTRY_CALL(CONTEXT_NAME: in STRING;
                         CHANNEL_NAME: in STRING;
                         TIME_LIMIT:   in DURATION :=
                             DURATION'LAST;
                         PARAMS: in PARAMS_STRING)
                         return     RESULTS_STRING;
        -- This routine sends the PARAMS to
        --  the designated context via the
        --  named channel.  It is delayed until
        --  the entry call is accepted, and
        --  the rendezvous is ended.
        -- Requires COMMUNICATE access over the
        --  designated channel,
        --  on the specified program context object.

procedure IPC_SELECT();
        -- IPC Select statement <<TBD>>

end INTER_PROGRAM_COMMUNICATION;
```

3.3.3.3.4.2 Processing

These interprogram communication primitives necessarily rely on the communicating programs agreeing on the format and interpretation of the PARAMS_STRING and RESULTS_STRING. From the KAPSE point of view, these are just character strings. A TIME_LIMIT of zero results in a conditional ACCEPT or ENTRY call. A TIME_LIMIT of DURATION'LAST (the default) results in an effectively un-timed call. If a single program wishes to receive ENTRY calls on many channels simultaneously, it may execute the

IPC_ACCEPT calls from separate Ada tasks, or use the <<TBD>> IPC_SELECT.

Certain channel names starting with an underscore (_CONTROL and _DEBUG) are reserved for the Ada Run Time System (KAPSE.RTS) and the Debugger Support Routines (see below).

3.3.3.3.4.3 Special Requirements

3.3.3.3.5 Debugging and Control Interface

3.3.3.3.5.1 Inputs and Outputs

The following procedures are available to a debugger for inspecting, controlling, and modifying a suspended program:

```
Package DEBUGGER_INTERFACE is

type PROGRAM_STATE is <<TBD>>;

procedure SET_CURRENT_DEBUGGED_CONTEXT(PROG_CTX: in STRING);
        -- This procedure is called once to specify
        --  which program context is being debugged,

procedure GET_PROGRAM_STATE(STATE: out PROGRAM_STATE);
        -- Retrieve the current state of the
        --  debugged program, including the program
        --  counter and stack pointer.

procedure CONTINUE(STATE: in out PROGRAM_STATE);
        -- Allow the debugged program to continue.
        -- This procedure returns when the debugged
        --  program reaches a breakpoint trap.

procedure SET_PROGRAM_DATA(ADDRESS: in ADDR_TYPE;
    DATA: in STORAGE_ARRAY);
        -- Store the array of storage units at the designated
        --  address in the debugged program.

procedure GET_PROGRAM_DATA(ADDRESS: in ADDR_TYPE;
    DATA: out STORAGE_ARRAY);
        -- Retrieve into the array of storage units
        --  from the designated address in the
        --  debugged program.

procedure SET_ECP_BREAKPOINT(ADDRESS: in ADDR_TYPE;
    ON_OFF: in BOOLEAN);
        -- Activate or deactivate a breakpoint at
        --  the designated execution control point,
        --  according to ON_OFF.

procedure SET_EXCEPTION_BREAKPOINT(EXCEPTION_ID: in INTEGER;
    ON_OFF: in BOOLEAN := TRUE);
        -- Associate or disassociate a breakpoint
        --  with the specified exception.

type BREAK_GROUP is (ALL_STATEMENTS,
                     ALL_CALLS,
                     ALL_EXCEPTIONS,
                     UNHANDLED_EXCEPTIONS,
                     ...<<TBD>>);

procedure SET_GROUP_BREAKPOINT(GROUP: BREAK_GROUP;
    ON_OFF: in BOOLEAN := TRUE);
        -- Associate or disassociate a breakpoint with
        --  the specified group of execution control
        --  points or exceptions.
```

```
end DEBUGGER_INTERFACE;
```

3.3.3.3.5.2 Processing

The above procedures are implemented using inter-program communication primitives. When a program is suspended, all of its normal tasks are made dormant, but a Debugger Support task remains responsive to inter-program communication on channel "_DEBUG." The Debugger Support task performs the requested operations on the debugger's behalf. See [AIE(1).DBUG(1)] for a more complete discussion of the debugging interface.

3.3.3.3.5.3 Special Requirements

3.3.3.4 KAPSE/KAPSE Communication

3.3.3.4.1 Inputs and Outputs

This CPC will provide a <<TBD>> interface for communication between KAPSEs on separate (virtural) machines. The interface will be as close as possible to the interface provided by KAPSE_PROGRAM_COMMUNICATION.

3.3.3.4.2 Processing on VM/SP

All communication between virtual machines is accomplished using the Inter-User Communication Vehicle or the Virtual Machine Communication Facility [IBM81]. Both of these methods provide an interrupt to the receiving VM when a message is ready. The data is copied using a fast memory to memory transfer.

3.3.3.4.3 Processing on OS/32

<<TBD>>

3.3.3.4.4 Special Requirements

3.3.3.5 Terminal Screen Manager

3.3.3.5.1 Inputs and Outputs

The KAPSE provides a standard set of terminal control facilities, directly available to the interactive MAPSE user:

ASCII Key Code	Terminal Control Function
Control-S (XOFF)	Stop terminal output. Enter Scroll Control Mode. (see below)
Control-Q (XON)	Exit Scroll Control Mode. Re-start terminal output.
Control-C (ETX) or BREAK	Interrupt running program, Give control to program catching input interrupts.
Control-H (Backspace)	Erase previous entered character.
Control-X (Cancel)	Erase entire line entered.

Scroll Control Mode is provided for terminal users to review output which has gone off the screen of a video terminal, or was illegible or lost from the printout of a hardcopy terminal.

In Scroll Control Mode, the terminal handler recognizes the following small number of commands:

ASCII Key Code	Scroll Control Mode Function
B	"Back" -- Scroll the screen backward half of a screen-ful, or simply retype the previous line on a hardcopy terminal.
digit B	Go back specified number of half screens or lines, and redisplay.
F	"Forward" -- Scroll the screen forward half of a screen-ful, or simply retype the next line which had been typed on a hardcopy terminal.
digit F	Go forward specified number of half screens or lines, and redisplay.

Control-C or BREAK	Exit Scroll Control Mode and Interrupt program as above.
Control-Q	Exit Scroll Control Mode, return to display of current terminal output.

On terminals without normal ASCII keyboards, the user may define alternate character sequences to replace the ASCII control characters. On half-duplex systems, all control characters (or sequences) must be preceded by an attention key, and terminated by the end-of-line character so that characters are received by the KAPSE.

3.3.3.5.2 Processing

Scroll Control Mode is possible because all terminal output is saved temporarily in the context object attribute 'TERMINAL_OUTPUT. At the end of program execution, this component may be saved if the output is considered valuable.

In addition, all terminal input to a program is stored temporarily in the context object attribute 'TERMINAL_INPUT, so that historical records of program invocation can be complete. At the end of program execution, a user may copy the 'TERMINAL_INPUT component into a more permanent part of the database to avoid having to re-enter the same input if the program is re-run at a later time. From the point of view of history, 'TERMINAL_INPUT is treated as a source object.

It is expected that the terminal handler will be enhanced to support multiple programs simultaneously on separate parts of the screen, with additional control characters for moving between the various screen windows.

3.3.3.5.3 Special Requirements

3.3.3.6 Login/Logout and User Context

3.3.3.6.1 Login/Logout and User Context Management

3.3.3.6.1.1 Inputs and Outputs

```
Package USER_CONTEXT is

procedure LOGIN(USER_NAME: in STRING;
    USER_PASSWORD: in STRING);
        -- This routine locates the user's
        --  top-level directory ("'ROOT.USERS." & USER_NAME),
        --  encrypts and compares the password,
        --  and then initiates the user's initial
        --  command processor.

procedure LOGOUT;
        -- This ends the current session, and
        --  logs the user out.

function CURRENT_USER_NAME return STRING;
        -- This function returns the current USER_NAME
        --  as specified to LOGIN.

procedure CHANGE_VIEW(PARTITION: in STRING);
        -- This procedure redefines the 'CURRENT_DATA
        --  window to refer to the newly selected
        --  PARTITION.
        -- It is implemented using standard window
        --  operations (i.e., CREATE_WINDOW)

procedure CHANGE_PASSWORD(PASSWORD: in STRING);
        -- This is meant to be suggestive.  Change
        --  password actually turns off echoing
        --  and requests the new password directly
        --  from the user's terminal.  After
        --  confirmation, the new password is
        --  stored as the value of the USER_PASSWORD
        --  attribute of "'TOP LEVEL_DATA." (see below).

end USER_CONTEXT;
```

3.3.3.6.1.2 Processing

When a user logs into the KAPSE, the LOGIN system requests a USER_NAME and a USER_PASSWORD (not echoed). The USER NAME is used to select a component from the USERS composite object. The password is encrypted using a non-invertible function and compared with the USER_PASSWORD attribute of this component. If the value matches, the component is taken to be the user's top-level directory (composite object), within which, by convention, exists an attribute named INITIAL PROGRAM_CONTEXT, which specifies what command processor is to be invoked on the user's behalf, with standard text input and output connected to the

user's terminal. The INITIAL_PROGRAM_CONTEXT normally identifies the executable program for a full command language processor, but may specify a more restrictive program designed to provide a user with a more controlled environment (e.g., text editing only).

No additional primitives are needed to manipulate the USERS composite object, or its components. Nevertheless, only users with an appropriate window on the USERS composite object can add new users to the system. Individual users may change their own USER_PASSWORD attribute, but not their USER_NAME.

When the MAPSE is initially installed, there is a single component of USERS named SYSTEM_MANAGER, with password SYSTEM. The SYSTEM_MANAGER composite object has an INITIAL_PROGRAM_CONTEXT with a SYSTEM window on the root of the entire database. The first action after installation should be to change the SYSTEM_MANAGER password.

Although a sophisticated user or project manager could create for themselves an arbitrary INITIAL_PROGRAM_CONTEXT (limited of course by their access rights), most users will choose to follow the APSE standard for program contexts attributes, which include the following:

Standard program-context attributes:

```
String Attributes          Value
-----------------          -----------------------------
PROGRAM_SEARCH_LIST   =>  "'TOOLS.,'CURRENT_DATA."

PARAMETERS            =>  ""
                              -- No parameters to top-level
                              --  context.
                      =>  "NAME=>ABC,COUNT=>3"
                              -- Example of parameters to
                              --  lower-level context.
RESULTS               =>  ""
                              -- Empty string while still
                              --  active.
                      =>  ",,NUM_ERRORS=>5"
                              -- Out parameters after
                              --  the program completes.
                      =>  "RETURN=>3.1415"
                              -- Result of completed
                              --  "function" program.
```

```
   CONTEXT_STATE          =>  "TERMINATED"
                                  -- State of program context
                                  --  after termination.
                          =>  "ABORTED"
                                  -- State of program context
                                  --  abnormally terminated.
                          =>  "COMPLETED"
                                  -- State of program context
                                  --  which has completed
                                  --  processing but which
                                  --  is waiting for its
                                  --  subcontexts to complete.

                          =>  "RUNNING"
                                  -- State of program context
                                  --  actively running.
                          =>  "SUSPENDED"
                                  -- State of program context
                                  --  suspended by user.
                                  -- Context is waiting for
                                  --  debugging commands,
                                  --  restart, or termination.
```

Window Attributes	Typical Target
'CURRENT_DATA	User's top-level composite object
'TOP_LEVEL_DATA	User's top-level composite object
'ROOT	Root of database
'TOOLS	TOOLS component of ROOT
'CALLER_CONTEXT	Context of invoker
'PROGRAM	The executable program object

Other Attributes	Value
'TERMINAL_INPUT	Simple text object
'TERMINAL_OUTPUT	Simple text object

```
        -- These two objects are managed by the
        --  KAPSE terminal handler.  Program
        --  I/O are connected to these text
        --  objects, with TERMINAL_INPUT lengthened,
        --  and TERMINAL_OUTPUT displayed
        --  by the terminal handler under
        --  keyboard control.
```

```
'OPEN_HANDLES          List of reserved handles
      -- Each open file (or partition) handle
      --  is represented by a reserved handle
      --  on the opened object (or partition),
      --  created within this list.
      -- OPEN_FILE HANDLES.1 and OPEN_FILE_HANDLES.2
      --  are always associated with standard text
      --  input and output, respectively.

'SUB_CONTEXT           Program context object
      -- This attribute is used by default to
      --  hold the context object for a program
      --  called as a sub-program.
      -- The PARAMETERS attribute of the context
      --  is the parameters to this sub-program.
```

The program context captures in one object the information the KAPSE needs to know about a running Ada program.

3.3.3.6.1.3 Special Requirements

The LOGIN procedure provides the primary protection against unauthorized access to the AIE. Therefore, it is required that the encryption algorithm of LOGIN be thoroughly tested for non-invertibility and for resistance to other code-breaking techniques.

3.3.3.6.2 User Accounting

<<TBD>>

3.3.3.7 Inter-User Mail System

3.3.3.7.1 Inputs and Outputs

The following programs are available for sending and receiving inter-user mail:

```
Package MAIL_SYSTEM is

procedure SEND_MAIL(TO_USER: in STRING; SUBJECT: in STRING;
    MESSAGE_OBJ: in STRING; MAIL_SEQ_NUM: out INTEGER);
        -- This program sends mail to the designated user.
        --  The program constructs a path as
        --  "^ROOT.USERS." & TO_USER & ".MAILBOX"
        --  and attempts to invoke the operation
        --  SEND on this private object.
        --  If the caller lacks sufficient access
        --  rights through this path, SEND_MAIL will fail.
        -- In addition, this requires a window allowing
        --  READ of the MESSAGE_OBJ.
        -- The returned MAIL_SEQ_NUM may be used to check
        --  if the mail has been read.

function SEND_MAIL_CHECK(TO_USER: in STRING;
    MAIL_SEQ_NUM: in INTEGER)
    return BOOLEAN;
        -- This function indicates whether the message
        --  with the specified MAIL_SEQ_NUM has been
        --  read.
        -- This function simply fails if the message
        --  was not sent by the caller.

function CHECK_MAIL return INTEGER;
        -- This function returns a count of the number
        --  of message objects in the user's MAILBOX.
        -- The path to the mailbox is assumed to be
        --  "^TOP LEVEL_DATA.MAILBOX"

procedure READ_MAIL(MESSAGE_OBJ: in STRING);
        -- The next message in the user's mailbox is
        --  is copied into the specified MESSAGE_OBJ.
        -- The following non-distinguishing attributes
        --  of this MESSAGE_OBJ will have appropriate values:

        --  FROM         =>   USER_NAME of SENDer,
        --  SUBJECT      =>   SUBJECT as specified by SENDer,
        --  MAIL_SEQ_NUM =>   Mail sequence number of this
        --                          message.

end MAIL_SYSTEM;
```

3.3.3.7.2 Processing

Mail is implemented using private object operations. When a new user is added to the system, the system manager creates a private object called MAILBOX in the user's top-level composite object by copying the standard system mailbox template. Each of the MAIL

subprograms given above simply invoke the appropriate operation of a mailbox private object.

For example, SEND_MAIL could be written in Ada as follows:

```
procedure SEND_MAIL(TO_USER: in STRING; SUBJECT: in STRING;
    MESSAGE_OBJ: in STRING; MAIL_SEQ_NUM: out INTEGER) is
        MAIL_PATH: constant STRING :=
                "^ROOT.USERS." & TO_USER & ".MAILBOX";
        MAIL_PARAMS: constant STRING :=
                "FROM_USER=>"    & CURRENT_USER_NAME &
                ",SUBJECT=>"      & SUBJECT              &
                ",MESSAGE_OBJ=>" & MESSAGE_OBJ;
begin
        MAIL_SEQ_NUM :=
          INTEGER'VALUE(PICK_PARAM(
            INVOKE_OPERATION(
                PRIV_OBJ     =>  MAIL_PATH,
                OPERATION    =>  "SEND",
                PARAMETERS   =>  MAIL_PARAMS
             ),
            "MAIL_SEQ_NUM"
          ));

end SEND_MAIL;
```

3.3.3.7.3 Special Requirements

3.3.4 History and Archiving (KAPSE.HISTARCH)

3.3.4.1 History and Archiving Operations

History records manipulations of objects, and provides for the reconstruction of previous states of objects.

3.3.4.1.1 Inputs and Outputs

History is recorded automatically while programs execute. History is made available to tools via the KAPSE/Tool interface package HISTORY:

```
with CALENDAR;  -- Defines type TIME.
Package HISTORY is

type HISTORY_CLASS is (SOURCE, DERIVED);
type HISTORY_REF(CLASS: HISTORY_CLASS := DERIVED) is private;
type HISTORY_REF_ARRAY is array(POSITIVE range <>) of HISTORY_REF;

function GET_HISTORY_REF(NAME: in STRING) return HISTORY_REF;
        -- get current "STATE" of object.

procedure RECREATE(STATE: in HISTORY_REF(CLASS=>SOURCE);
    NAME: in STRING);
        -- Given the "STATE" of a source object, recreate
        --  its content and user attributes in a new database
        --  object with the given NAME.

procedure NEW_SOURCE_ARCHIVE(SOURCE_OBJ: in STRING);
        -- This creates a new source archive with
        --  SOURCE_OBJ as its state number one.

procedure OLD_SOURCE_ARCHIVE(SOURCE_OBJ: in STRING;
    STATE: in HISTORY_REF(CLASS=>SOURCE));
        -- This specifies that SOURCE_OBJ is a
        --  revision of STATE, and should be
        --  assigned to the same source archive.

function GET_DIRECT_CONSTITUENTS(STATE: in HISTORY_REF)
    return HISTORY_REF_ARRAY;
        -- Given STATE, return list of states from which
        --  this state was directly derived.  If object is
        --  a source object, no more than one state is
        --  returned -- that of the direct predecessor
        --  to this state.

function GET_SOURCE_CONSTITUENTS(STATE: in HISTORY_REF)
    return HISTORY_REF_ARRAY;
        -- Given STATE, return list of source states from
        --  which this state was derived, directly or
        --  indirectly.  Derived object states are included
        --  in list only if their history was off-line
        --  and thus could not be traced immediately.

function GET_HISTORY_PARAMETERS(STATE: in HISTORY_REF)
    return STRING;
        -- For derived object state, return STRING
        --  with parameters provided at
        --  invocation of program producing STATE.
        -- For source object state, return list
        --  of the user attributes of the object
        --  at time of merge into archive.
        -- STRING is returned in (label=>value,...) format.

procedure HISTORY_ACTIVATE(STATE: in HISTORY_REF;
```

```
        TIME_LIMIT: in DURATION);
            -- This procedures requests that a particular history
            --  script or archive be activated (brought on-line).
            -- Depending on bulk-storage hardware, this may occur
            --  immediately or await operator attention, up to the
            --  specified TIME LIMIT.

    function HISTORY_ON_LINE(STATE: in HISTORY_REF) return BOOLEAN;
            -- This function returns TRUE if the referenced history
            --  script or archive is now active (on-line).

    function HISTORY_TIME(STATE: in HISTORY_REF)
        return CALENDAR.TIME;
    function HISTORY_MAKER(STATE: in HISTORY_REF)
        return STRING;
            -- The above two functions return the time/date and
            --  USER_NAME associated with the specified script
            --  or source archive STATE.
```

3.3.4.1.2 Processing

The history attribute of a database object represents its "state," and consists of a an index and a window on either a source archive for a source object, or a program invocation script for a derived object (see 3.2.4.3.9). The index is used to select one state from all the states associated with the same source archive or script.

Scripts and archives are exteneded objects created within the SYSTEM component of the ROOT composite object, with attributes to indicate whether the content is present, or has been moved off-line to tape. When the script or source archive is moved off-line, its content is copied to tape and then deleted, and its attributes are set up to identify which tape holds the data.

If the referenced history is off-line, many of the above primitives will fail. The primitives HISTORY_ACTIVATE and HISTORY_ON_LINE may be used to affect or check the on-line status of a particular source archive or script.

All objects when initially created are treated as derived objects, with a HISTORY that refers to a program invocation script. The primitives NEW_SOURCE_ARCHIVE and OLD_SOURCE_ARCHIVE may be used to replace the HISTORY attribute's window on the script by a read-only window on a source archive. Source archives are used for maintaining multiple states of the same basic text, where the content itself is more important than the script of the program invocation used to create the content. The date, time, and USER_NAME from the program invocation script are

transferred to the source archive for each of its component states.

The source archive is stored in a form allowing the efficient reconstruction of any of the component states, while minimizing the space necessary to represent the multiple states.

History scripts for derived objects are created automatically while programs execute. The program invocation script first records the date, time, USER_NAME, and parameters. The count of modified objects is initialized to zero. As any database object is opened/created for reading or writing, a read-only window is entered in the script referring to the object, plus a copy of its pre-existing history attribute (if any). When an object which has been modified is closed, the count of modified objects is incremented, and the object's history attribute is updated to include a read-only window on the script, and the current modification count as the history index.

3.3.4.1.3 Special Requirements

The space occupied by history scripts and source archives, as well as the time required to record them (for scripts), or insert in/extract from them (for archives) must be as small as possible to preserve overall performance of the KAPSE.

3.3.4.2 Backup and Recovery

An important design feature of the KAPSE is that backup and incremental recovery can be performed while the system is up and running. The tape (or bulk-storage) backup program begins by simply doing a READ_COPY reserve of the root of the entire database. After that operation, the backup program may progress at its own pace through the hierarchy of objects, knowing that the data it reads reflects an internally consistent snap shot of the entire database.

3.3.4.2.1 Inputs and Outputs

The following system programs are available for full and incremental backup, and incremental recovery:

```
Package BACKUP_RECOVERY is

procedure FULL_BACKUP(TIMESTAMP: out TIME_SEQ_NUMBER);
        -- This program copies a snapshot of the entire
        --  database to the tapes mounted by the operator.
        -- TIMESTAMP is the maximum time sequence number
        --  of any of the blocks transferred to tape.

procedure INCREMENTAL_BACKUP(BASE_LINE: in TIME_SEQ_NUMBER;
    TIMESTAMP: out TIME_SEQ_NUMBER);
        -- This program copies blocks to tape that have been
        --  modified since the BASE_LINE time sequence number.
        -- It also copies any block superior to a block that
        --  has been modified, to ensure that the copy on
        --  tape is a connected DAG (directed acyclic graph).

procedure RECOVERY(OLDNAME: in STRING; NEWNAME: in STRING;
    TIMESTAMP: in TIME_SEQ_NUMBER);
        -- This program attempts to re-create as NEWNAME
        --  the specified object as it was at the specified
        --  time sequence number.

 ...

end BACKUP_RECOVERY;
```

3.3.4.2.2 Processing

The KAPSE maintains an index of all backup tapes, indicating the range of time sequence numbers appearing on the tape. Each backup tape includes a header identifying its range. The rest of the tape is in a standard format with each block including its BLOCK_ID and reference count from when the block was dumped from disk. The blocks are topologically sorted before being dumped so that any element of the hierarchy on the tape may be located in a single sequential scan through the tape.

On recovery, the KAPSE instructs the operator to mount the appropriate incremental and full backup tapes, in order from latest to earliest, until the full content of the specified object has been reconstructed as of the requested time sequence number.

3.3.4.2.3 Special Requirements

3.3.4.3 Configuration Management Support

Configuration reporting and management are not separable from the rest of the KAPSE database facilities, but are rather integral to the reporting and management of attributes and partitions. The following KAPSE primitives, described in other sections of this document, are particularly relevant:

KAPSE Primitive	Section of this document
SET_ATTRIBUTE GET_ALL_ATTRIBUTES	3.3.2.2
CREATE_WINDOW	3.3.2.1
OPEN_PARTITION GET_NEXT_COMPONENT	3.3.1.5
SET_ROLE_ACCESS GET_ROLE_ACCESS GET_ROLES	3.3.2.3.1
GET_DIRECT_CONSTITUENTS GET_SOURCE_CONSTITUENTS GET_HISTORY_REFS	3.3.4.1

In addition to the above primitives, a small set of standard MAPSE tools are provided to exemplify the use of the facilities.

3.3.4.3.1 Partition Listing Tool

3.3.4.3.1.1 Inputs and Outputs

This tool is designed to produce the configuration and attribute reports required by the [SOW80]:

```
procedure LIST_PARTITION(PARTITION: in STRING := "'CURRENT_DATA.";
    ATTRIBUTES: in STRING := "");
        -- This program prints on the standard text
        --  output the distinguishing attributes
        --  (ie., names) of all of the components of the
        --  specified partition, as well as the requested
        --  non-distinguishing attributes, specified in
        --  the parameter ATTRIBUTES as a
        --  comma-separated list of attribute labels.
        -- If ATTRIBUTES is "*" then all non-null
        --  attributes of the components are printed.
        -- If ATTRIBUTES is null then no non-distinguishing
        --  attributes are printed.
        -- Notice that by default, the program lists only the
        --  distinguishing attributes of all of the components
        --  of the partition implied by the .CURRENT DATA
        --  window.
```

This program may be used to list attributes of:

1. The components of a composite object (ie., a configuration);

2. Some subset of the components, which satisfy a more complicated partition specification;

3. A single simple object.

3.3.4.3.1.2 Processing

The program LIST_PARTITION is implemented using the KAPSE primitives OPEN_PARTITION, GET_NEXT_COMPONENT, and GET_ALL_ATTRIBUTES.

3.3.4.3.1.3 Examples

```
SET_ATTRIBUTE("ALPHA", "PURPOSE", "FUN");
SET_ATTRIBUTE("ALPHA", "CHECK_LEVEL", "2");
SET_ATTRIBUTE("BETA", "PURPOSE", "WORK");
SET_ATTRIBUTE("BETA", "CHECK_LEVEL", "2");
SET_ATTRIBUTE("GAMMA", "PURPOSE", "FUN");

LIST_PARTITION("(CHECK_LEVEL=>2)", "PURPOSE");
    -- The following would appear on the standard output:
    --
    --  Partition (CHECK_LEVEL=>2)      Attributes PURPOSE
    --  ALPHA                           PURPOSE=>FUN
    --  BETA                            PURPOSE=>WORK

LIST_PARTITION("(PURPOSE=>FUN)", "CHECK_LEVEL");
    -- The following would appear:

    --  Partition (PURPOSE=>FUN)        Attributes CHECK_LEVEL
    --  ALPHA                           CHECK_LEVEL=>2
    --  GAMMA                           No CHECK_LEVEL

LIST_PARTITION;  -- Use the defaults
    -- The following might appear:

    --  Partition 'CURRENT_DATA.
    --  ALPHA
    --  BETA
    --  DELTA
    --  GAMMA
    --  KAPPA

    -- Notice that all partitions are sorted in ASCII
    --  lexicographic order.
```

3.3.4.3.1.4 Special Requirements

3.3.4.3.2 A Configuration Management Facility

3.3.4.3.2.1 Inputs and Outputs

This set of tools provides a simple configuration management facility:

```
function  MOST_RECENT(PARTITION: STRING; ATTRIBUTE_LABEL: STRING)
    return STRING;
        -- This program scans the designated partition for the item
        --  with the largest value for the specified
        --  attribute, and returns that value as a string.
        -- The presumption is that the designated attribute is
        --  being used as a revision number, and the desire is
        --  to determine the most recent revision.

procedure ITEM_RESERVE(ITEM_NAME: STRING; WINDOW_PATH: STRING);
        -- This tool reserves the item with the specified name,
        --  and creates a window at WINDOW_PATH through which
        --  the item can be created/edited as necessary.
        -- The ITEM_NAME may be the name of an existing object,
        --  or it may be a name generated by determining
        --  the MOST_RECENT revision and incrementing the revision
        --  attribute value to create a new name, or by assigning
        --  it a new value for some version-like distinguishing
        --  attribute.
        -- A copy of the state of the object is made for fallback
        --  on ABORT (see below).
        -- This program will exit with an error if the ITEM is
        --  already reserved, or the user does not have rights
        --  to reserve it.

procedure ITEM_RELEASE(WINDOW_PATH: STRING);
        -- This tool deletes/revokes the designated window, and
        --  releases the item associated with it for access by
        --  others via ITEM_RESERVE.  The fallback copy is deleted.

procedure ITEM_ABORT_RESERVE(WINDOW_PATH: STRING);
        -- This tool aborts the reservation of the associated
        --  item, and restores it to the fallback state.

function  WHO_HAS_IT(ITEM_NAME: STRING) return STRING;
        -- This tool returns the USER_NAME of
        --  the user who performed the reserve on the item,
        --  or returns the null string if the item is not
        --  reserved.
```

3.3.4.3.2.2 Processing

The processing of these tools can all be quite easily defined using the primitives of the KAPSE/Tool interface, especially those primitives identified at the beginning of this section. In general, they create and delete windows on the partition which includes exactly the item being reserved, and record in user-defined non-distinguishing attributes of the item the fact that it is reserved, and a fall-back copy of it.

Note that this notion of "RESERVE" persists across program executions, while the RESERVE of the KAPSE/Tool interface package

ACCESS_SYNCHRONIZATION (3.3.2.3) only applies while a program is running, and is automatically released when the program exits. Nevertheless, the primitives provided by the KAPSE are essential to safely implementing a persistent reservation, if only to ensure that two independent programs do not accomplish the ITEM_RESERVE simultaneously.

3.3.5 Run-time System (KAPSE.RTS)

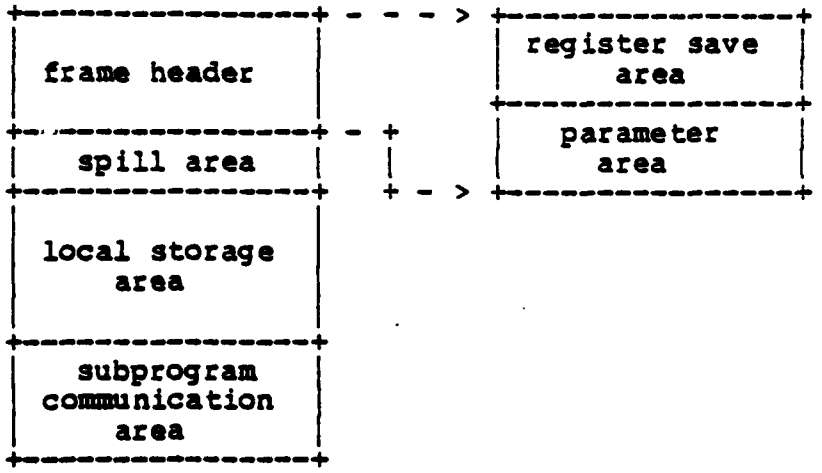
3.3.5.1 Unit Execution Support

The following sections describe the techniques used to support the execution of Ada subprograms and blocks on IBM 4341 VM/SP systems. The subprogram calling conventions used are not compatible with existing IBM 4341 conventions; special interface coding will be required if subprograms compiled by other language processors are to be called.

3.3.5.1.1 Inputs

3.3.5.1.1.1 Call Frames

A call frame is a contiguous block of storage, normally allocated on the primary stack, which contains the saved registers, parameters, and static-sized local variables associated with a particular subprogram or entry invocation. A typical call frame is laid out as follows:



The fields in a call frame contain the following information:

1. Frame Header. A frame header contains a register save area and parameter area. The register save area is used to store the contents of general registers which must be preserved across the execution of the unit. These registers are saved on entry and restored on exit. The parameter area contains actual parameter values and references. The size and layout of the parameter area is dependent on the number and type of formal parameters specified for the subprogram.

2. Spill Area. The spill area is used by the generated code to temporarily store the contents of registers when they are needed for other purposes. The size of the spill area needed is statically determined by the compiler.

3. Local Storage Area. The local storage area contains all statically-sized (i.e. those whose size is known at compile time) local variables. The local storage area also contains pointers to dynamically-sized local variables for which space has been allocated on the secondary stack.

4. Subprogram Communication Area. The subprogram communication area is required in a call frame if any non-static subprograms are called from within the body of the current subprogram. The communication area is used to store the frame headers for these called subprograms, and has a pre-allocated size sufficient to contain the largest of the headers.

3.3.5.1.1.2 Parameter Passing

The method used to pass a parameter is dependent on the type declared for the formal:

1. Scalar or access. Passed by copy.

2. Constrained record or array. If the values are 8 bytes or less in length, pass by copy, otherwise pass by reference (address).

3. Unconstrained record. Passed by reference. If the actual parameter value is constrained, the caller sets a flag in the reference, the formal parameter then inherits the constraints which applied to the actual.

4. Unconstrained array. Passed by reference. A descriptor for the array must be provided by the caller, a reference to the descriptor is passed as an additional implicit parameter.

5. Task. Passed by reference (address of TCB).

The parameters are passed in the parameter area, with those needing double-word alignment (long floating values) first, full-word alignment (references, access values, integers, etc.) second, and half-word alignment (enumeration, short integer values) last.

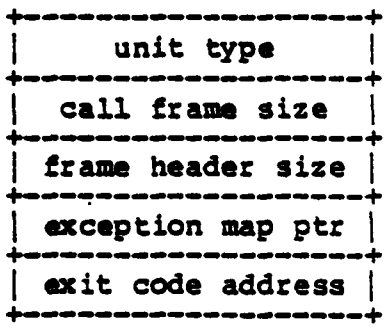
Scalar values less than 16 bits in length (enumeration and boolean types) are passed right-justified and zero padded in a 16 bit half-word.

References consist of a 24 bit memory address right justified in a 32 bit full-word. References to unconstrained

record types contain a constrained flag in the leftmost (sign) bit of the word.

3.3.5.1.1.3 Unit Data Area

Each subprogram body is preceded by a unit data area which contains static information specific to that body. The unit data area contains the following fields:



The executable code for a unit begins at a fixed (for all units) offset from the beginning of its unit data area.

3.3.5.1.1.4 Register Usage

The operations provided use the following registers for passing parameters:

reg.	normal use
CODE	Code base register.
GLBL	Global base register.
FRAME	Pointer to current call frame.
RET	Subprogram return address.
SBC	Static back chain.
UDA	Pointer to unit data area for current unit.
SCA	Pointer to subprogram communication area in current call frame.

3.3.5.1.1.5 Execution Support Operations

The following operations are provided to support the execution of executable units:

oper.	parameters	description
SAVREG	SCA, FRAME, UDA	Save caller's registers.
RSTREG	FRAME	Restore caller's registers.
SUBCALL	UDA, SBC, SCA, RET	Call subprogram.
GENPASS	UDA, SBC, SCA, RET	Pass generic subprogram parameter.
GENCALL	UDA, SBC, SCA, CODE, RET	Call generic subprogram parameter.

3.3.5.1.2 Processing

3.3.5.1.2.1 Subprogram Calls

SUBCALL is used to call subprograms which follow the standard Ada calling conventions. Prior to executing a SUBCALL, UDA must contain the address of the unit data area for the subprogram, and SBC contains the address of the call frame belonging to the static parent of the subprogram (when needed). SUBCALL loads the address of the instruction following the SUBCALL into RET, and branches to the body of the indicated subprogram (which is at a fixed offset relative to UDA).

All general registers, other than the designated temporaries, are preserved across a SUBCALL. All of the floating point registers are considered to be temporary, so they must be saved by the caller as needed.

3.3.5.1.2.2 Subprogram Prologue Code

Prologue code precedes the code which actually implements the body of a subprogram. The standard prologue executes SAVREG to save the caller's general registers in the caller's subprogram communication area. ALFRAME (see Storage Management) is then called to allocate the subprograms call frame on the primary stack. Following the prologue code, FRAME contains the address of the call frame, and CODE has been loaded with the base address of the first subprogram code section.

3.3.5.1.2.3 Subprogram Code Sections

A subprogram body consists of one or more code sections. Each code section is a maximum of 4096 bytes long and consists of a series of instructions followed by the literal pool for that section. Base register CODE always points to the beginning of the current code section, and must be set up prior to entering a new section.

3.3.5.1.2.4 Subprogram Exit Code

The normal exit code executes RSTREG to restore the caller's registers, then returns control to the next instruction in the caller's code (which is pointed to by RET). A call to RLSTKSEG is automatically executed if the subprograms call frame was the last one in the current stack segment.

The address of the normal exit code is stored in the unit data area so that it will be available to the exception handling mechanism.

3.3.5.1.2.5 Static Subprograms

The call frame for a static subprogram is allocated in static storage rather than on the primary stack. The caller stores the actual parameters directly in the preallocated frame. When calling a static subprogram, SCA must be loaded with the address of this frame; the calling sequence is otherwise identical to that for normal subprograms.

In addition to the statically allocated frame, a static subprogram requires a vestigial call frame on the primary stack, consisting only of a subprogram communication area, if normal subprograms are called from within the body. If no such calls are made, the code to allocate a frame on the stack may be omitted from the prologue.

3.3.5.1.2.6 Generic Subprogram Parameters

Ada permits entries or procedures to be used interchangeably as actuals for generic formal subprogram parameters (see Generic Instantiation). Inside an instantiation, it is not possible to distinguish between the two cases. Two operations are provided permit transparent implementation of calls to generic subprogram parameters. GENPASS is executed to pass an actual entry or procedure to a newly instantiated generic. Prior to executing GENPASS, UDA, SCA, and SBC must be loaded with the information needed when the actual entry or procedure is called. GENPASS stores the contents of these registers in the generic subprogram parameter descriptor corresponding to the parameter.

GENCALL is executed to actually call the parameter. UDA, SCA, and SBC are reloaded from the parameter descriptor, and control is passed to the actual procedure or entry.

3.3.5.1.3 Outputs

A function subprogram returns a scalar value in one of the following registers:

VAL Integer, enumeration, fixed point or access values.
FPVAL Floating point values.

Non-scalar values are returned as follows:

1. Constrained array or record. Space for the result is preallocated by the caller, address passed to function via an implicit parameter.

2. Unconstrained record. Result allocated by function on secondary stack, address returned to caller via an implicit parameter.

3. Unconstrained array. Same as unconstrained record types, in addition, array descriptor is returned to caller using a second implicit parameter.

3.3.5.1.4 Special Requirements

Due to the special interfaces required and to achieve a reasonable level of efficiency, all unit execution support operations will be implemented in IBM 4341 machine language.

3.3.5.2 Storage Management

The following run-time storage structures are used to support the execution of compiled Ada code:

1. Storage Segment. A storage segment is a contiguous block of memory which is a multiple of PAGESZ (a compiler parameter) bytes in length. Storage segments are used to implement stacks and heaps.

2. Primary Stack. Primary stacks are used exclusively for the storage of call frames (see Unit Execution Support) following normal stack discipline. Each task in a program (and the main subprogram) has a primary stack associated with it.

3. Secondary Stack. Secondary stacks are used for the storage of local variables and function return values whose size could not be determined at compile-time. A secondary stack is managed using a mark/release strategy. A main program and each of the executable tasks within the program has a secondary stack associated with it.

4. Collections. A collection is used for the storage of access data (i.e. data referenced by access values) associated with an access type which was defined with a STORAGE_SIZE clause. Storage for individual data items within a collection can not be reclaimed; the storage occupied by the entire

collection is reclaimed on exit from the unit in which the type was declared.

5. Controlled Heap. Access data belonging to a type for which a CONTROLLED pragma has been supplied is allocated on the controlled heap. Controlled heap data is not automatically reclaimed; the user may explicitly deallocate such data using an UNCHECKED_DEALLOCATION procedure which has been instantiated for the type.

6. Checkpoint Heap. Access data belonging to a type for which a MARK_RELEASE pragma has been supplied is allocated on the checkpoint heap. The user may mark the checkpoint heap using the supplied MARK procedure. The user may at some later time call the RELEASE procedure to reclaim storage allocated after the corresponding call to MARK.

7. Default Heap. For an access type, the user may specify at most one of the three preceding storage categories (through a STORAGE_SIZE clause, or a CONTROLLED or MARK_RELEASE pragma). If none of these categories is specified, data is allocated on the default heap. The default heap is allocate-only, storage can not be reclaimed through user action or automatically.

3.3.5.2.1 Inputs and Outputs

The storage management package implements a variety of low-level operations which are invoked by the generated code. A user visible package is also provided which allows the user to mark and subsequently release storage on the checkpoint heap.

3.3.5.2.1.1 Register Usage

The low-level operations use the following registers for passing parameters:

reg.	normal use
SIZE	Size of object (in bytes).
PTR	Pointer to object (address).
VAL	Function return value.
DESC	Pointer to collection descriptor.
FRAME	Pointer to current call frame.
SCA	Pointer to subprogram communication area in current call frame.
UDA	Pointer to unit data area for current unit.

3.3.5.2.1.2 Primitive Storage Operations

oper.	registers	description
ALPAGES	SIZE, PTR	Allocate a block of memory pages.
ALSEG	SIZE, PTR	Allocate a storage segment.
RLSEG	SIZE, PTR	Release (deallocate) a storage segment.

3.3.5.2.1.3 Stack Operations

oper.	registers	description
ALFRAME	SCA, UDA, FRAME	Allocate a call frame on primary stack.
RLSTKSEG	SCA, FRAME	Release a primary stack segment.
MKSECSTK	PTR	Mark secondary stack.
ALSECOBJ	SIZE, PTR	Allocate object on secondary stack.
RLSECSTK	PTR	Release secondary stack storage.
CPSECSTK	VAL, PTR, SIZE	Copy secondary stack object.

3.3.5.2.1.4 Heap Operations

oper.	registers	description
ALCOLOBJ	DESC, SIZE, PTR	Allocate object in collection.
ALCTLOBJ	SIZE, PTR	Allocate object on controlled heap.
RLCTLOBJ	SIZE, PTR	Release controlled heap object
ALCHKOBJ	SIZE, PTR	Allocate object on checkpoint heap.
ALOBJECT	SIZE, PTR	Allocate object on default heap.

3.3.5.2.1.5 User Checkpoint Heap Operations

```
package MARK_RELEASE is
        type CHECKPOINT is limited private;
        exception RELEASE_ERROR;
        procedure MARK (CP: out CHECKPOINT);
        procedure RELEASE (CP: CHECKPOINT);
end MARK_RELEASE;
```

3.3.5.2.2 Processing

3.3.5.2.2.1 Primitive Operations

A two-level storage management scheme will be used to minimize the effects of storage fragmentation. The operations discussed in this section manipulate storage segments, which are a fundamental storage structure used to implement both stacks and heaps. Storage segments are always allocated in increments of the host machines page size (4096 bytes on the IBM 4341). A pool of free storage segments is maintained on a program wide basis. When this pool is exhausted, additional memory is obtained directly from the host operating system.

ALPAGES is called to obtain a contiguous block of virtual memory pages. On entry, SIZE is expected to contain the size of the block needed (which should be a multiple of PAGESZ bytes). The block of memory is obtained through a request to the host operating system (VM/SP). The base address of the block is returned in PTR.

ALSEG is called to obtain a storage segment. On entry, SIZE is expected to contain the size of the segment (which should be a multiple of PAGESZ bytes). If the free segment pool contains a segment of at least the requested size, then that segment will be returned. Otherwise, ALPAGES is called to obtain additional memory. If ALPAGES is unable to obtain additional memory, STORAGE_ERROR is raised. The base address of the segment is returned in PTR.

RLSEG is called to release a storage segment which is no longer in use. On entry, SIZE must contain the size of the segment (which must be the value specified when the segment was allocated), and PTR must contain the base address of the segment. The segment is returned to the free segment pool.

3.3.5.2.2.2 Stack Operations

A primary stack consists of zero or more, not necessarily contiguous, storage segments. Call frames are normally allocated on the primary stack on entry to subprograms, accept bodies, and other executable units. A call frame must fit entirely within a single stack segment to avoid addressing problems.

ALFRAME is called to allocate a call frame on the current primary stack. On entry, SCA is expected to contain the address of the subprogram communication area (see Unit Execution Support) in the call frame of the calling unit, and UDA should contain the address of the unit data area for the current (called) unit. The size of the stack frame needed is obtained from the unit data area. The new frame is allocated starting at the base of caller's communication area. If there is not sufficient room in the current segment to allocate the frame, a new segment is obtained by calling ALSEG, the caller's communication area is copied to the beginning of that segment, and the frame is then allocated. The base address of the frame is returned in FRAME.

Call frames are implicitly deallocated on exit from the units where they were allocated. When the last frame in a segment is deallocated, RLSTKSEG is called to return the segment to the free pool (through a call to RLSEG).

A secondary stack consists of zero or more storage segments. It is normally marked on entry to, and released on exit from, executable units and blocks which contain dynamically-sized local variables and do not return a dynamically-sized result. An

object allocated on the secondary stack must reside entirely within a single storage segment.

MKSECSTK is called to obtain the current secondary stack mark. On entry, PTR must contain the address of a location which will receive the mark. Pointers to the current secondary stack segment and the next available byte within that segment are stored at that location.

ALSECOBJ is called to allocate an object on the current secondary stack. On entry, SIZE must contain the size of the object in bytes. If there is sufficient room in the current secondary stack segment, then the object is allocated there. Otherwise, a new secondary stack segment is obtained through a call to ALSEG, and the object is allocated at the beginning of that segment. The address of the allocated object is returned in PTR.

RLSECSTK is called to release the storage allocated on the current secondary stack since an earlier call to MKSECSTK. On entry, PTR must contain the address of the location where the mark was stored. Secondary stack segments allocated since the mark was set are released through calls to RLSEG, then the secondary stack pointers are reset specified by the mark.

A special case in Ada is the declaration of a constant whose size is known only after the initialization expression is evaluated. Since space for the constant can not be allocated on the secondary stack before the expression is evaluated, CPSECSTK is called after evaluation to copy the constant to a mark which was set prior to evaluation. On entry, SIZE must contain the actual size of the constant, VAL must contain the address at which the constant was left following evaluation, and PTR must point to the location of a secondary stack mark which was set before evaluation. If there is sufficient room in the secondary stack segment indicated by the mark, the constant is copied there. Otherwise, a new secondary stack segment is obtained and the constant is copied to it. Any unused segments in the secondary stack following the mark are then released through calls to RLSEG. The new address of the constant is returned in VAL.

3.3.5.2.2.3 Heap Operations

The allocation operations described in this section are used to implement allocators (new operators) for the corresponding categories of access types.

A fixed amount of space is allocated for a collection when the access type definition is elaborated. The space may be allocated in static storage, in a call frame, or on the secondary stack, depending on where the type was declared, and whether the

size of the collection could be statically determined. A collection descriptor is constructed at the beginning of the allocated space. This descriptor contains an allocation pointer, which initially points to the first free location within the collection, and a pointer to the end of the collection.

ALCOLOBJ is called to allocate an object within a collection. On entry, DESC must contain the address of the collection descriptor, and SIZE must contain the size of the object in bytes. If sufficient space remains in the collection, the object is allocated, and its address is returned in PTR. Otherwise, STORAGE_ERROR is raised.

An UNCHECKED_DEALLOCATION procedure may be instantiated for an access type which is implemented using a collection. The only effect is to clear the specified access variable.

The controlled heap is implemented using zero or more storage segments. The size of the controlled heap may be expanded through allocation of additional segments, however, these segments are never reclaimed. A pool of the free objects within the segments is maintained for the entire program.

ALCTLOBJ is called to allocate an object in the controlled heap. On entry, SIZE must contain the size of the object in bytes. If there is an object in the free pool of at least the specified size, that object is removed from the free pool. Otherwise, a new segment is obtained by calling ALSEG, the object is allocated within that segment, and any excess space is added to the free pool. The address of the object is returned in PTR.

RLCTLOBJ is called to deallocate an object previously allocated using ALCTLOBJ. On entry, SIZE must contain the size of the object (which must be the value specified when the object was allocated), and PTR the address of the object. The object is returned to the free object pool.

Instantiation of an UNCHECKED_DEALLOCATION procedure for a controlled access type results in the generation of a routine which calls RLCTLOBJ to deallocate the specified object, then clears the access variable.

The checkpoint heap is implemented using zero or more storage segments. Storage allocated to the checkpoint heap may be explicitly reclaimed by the user through calls to the procedures provided in package MARK_RELEASE, which is described in the next section.

ALCHKOBJ is called to allocate an object in the checkpoint heap. On entry, SIZE must contain the size of the object in bytes. If there is sufficient room in the current checkpoint heap segment, the object is allocated there. Otherwise, a new segment for the checkpoint heap is obtained through a call to

ALSEG, and the object is allocated at the beginning of that segment. The address of the object is returned in PTR.

Instantiation of an UNCHECKED_DEALLOCATION procedure for a checkpoint access type results in the generation of a routine which simply clears the specified access variable.

The default heap is implemented using zero or more storage segments. Space allocated for the default heap can not be reclaimed. ALOBJECT is called to allocate an object in the default heap. On entry, SIZE must contain the size of the object in bytes. If there is sufficient room in the current default heap segment, the object is allocated there. Otherwise, a new segment is allocated for the default heap, and the object is allocated at the beginning of that segment. The address of the object is returned in PTR.

Instantiation of an UNCHECKED_DEALLOCATION procedure for a default access type results in the generation of a routine which simply clears the specified access variable.

3.3.5.2.2.4 User Checkpoint Heap Operations

Package MARK_RELEASE is visible to the user, and the procedures contained within may be called to manage the storage associated with the checkpoint heap. MARK is called to obtain the current checkpoint heap mark. The parameter must be a variable of type CHECKPOINT. Pointers to the current checkpoint heap segment and the next available byte within that segment are stored in the variable.

RELEASE is called to reclaim the storage occupied by checkpoint heap objects allocated since the specified CHECKPOINT variable was set. Any checkpoint heap segments which are no longer needed are released through calls to RLSEG, and the checkpoint heap pointers are reset to the values indicated in the variable. If the variable does not contain a valid mark within the checkpoint heap, RELEASE_ERROR is raised.

3.3.5.2.3 Special Requirements

Due to the special interfaces required and to achieve a reasonable level of efficiency, all storage management operations will be implemented in IBM 4341 machine language.

3.3.5.3 Tasking Support

The AdaTasking package implements Ada tasking operations by providing a number of types, objects, and low-level operations to compiled code. Compiled Ada code executes each Ada tasking

construct by calling one or more of the operations in the package. The AdaTasking package also contains internal types, objects, and operations which support the implementation of tasking operations, but are not available outside the package. These internal operations perform functions such as queue and list management, task scheduling, and context-switching between tasks.

Ada identifiers defined in this section are given in mixed upper and lower case. Machine language identifiers, and Ada identifiers defined in language standard packages are given in upper case.

3.3.5.3.1 Inputs

Interfacing between compiled code and the AdaTasking package utilizes the following types of data, in addition to items described in Special Calling Sequences:

A list of dependent tasks for each master, and lists of unactivated tasks for allocators:

```
type TaskListType is private;
type TaskListPtr is access TaskListType;
```

A task type descriptor for each task type:

```
type TaskTypeDescriptor is private;
type TaskTypePtr is access TaskTypeDescriptor;
```

A task Control Block for each task:

```
type TaskControlBlock is private;
type TCBptr is access TaskControlBlock;
```

A unique index (1..MaxEntry) for each simple entry and each member of an entry family in a task:

```
type EntryIndex is private;
```

Unit Data Areas for each task body's code, for each ACCEPT body's code, and for each entry:

```
type UnitDataArea is private;
type UDAptr is access UnitDataArea;
```

Task priorities, from PRAGMA PRIORITY statements:

```
type PriorityNumber is private;
```

Delays and time limits:

```
type DURATION is private; -- In package STANDARD.
```

Interrupts:

```
type InterruptType is private;
```

Work spaces for certain tasking operations:

```
type SelectRecord is private;
```

3.3.5.3.1.1 Task Body Unit Data Area

Each task body may contain some executable code. Associated with this code is a Unit Data Area (UDA), which describes the stack frame, exception map, and starting code address for the task body. The format for this Unit Data Area is identical to the Unit Data Area for subprograms (see Unit Execution Support).

3.3.5.3.1.2 ACCEPT Body Unit Data Area

Each simple ACCEPT statement and each ACCEPT alternative of a SELECTIVE WAIT may have code in an ACCEPT body. Associated with this code is a Unit Data Area, which describes the stack frame, exception map, and starting code address for the ACCEPT body. The format for this Unit Data Area is identical to the Unit Data Area for subprograms (see Unit Execution Support).

3.3.5.3.1.3 Entry Unit Data Area

Associated with each entry in a task type is an additional Unit Data Area, which specifies the stack space required for the entry's parameters. The tasking package uses this UDA for any entry call for which no corresponding ACCEPT has yet been executed. In addition to the Unit Data is a unique integer index for the entry. In the case of entry families, it is the index preceding the first family member.

3.3.5.3.1.4 Task Control Block

Each task has a task control block (TCB), and is identified by a pointer to that TCB. The TCB contains run-time information about the task, including:

Task status,

Delay information,

Code and stack context (when not running),

Links on queues and lists,

The set of currently open entries,

Information for each entry:
    The queue of callers waiting on the entry,
    If the entry is open, a UDA pointer to the ACCEPT body.

3.3.5.3.1.5 Dependency List

All dependent subtasks of a master (subprogram, task, block, or library package) are kept on a dependency list for that master. The tasking package uses this list to determine when tasks can be terminated, whether a block can be exited, and which subtasks are affected by an ABORT statement.

3.3.5.3.1.6 Special Calling Sequences

All AdaTasking operations not listed in this section are called from compiled code with the normal subprogram calling sequence (see Unit Execution Support). The following operations require special calling sequences from compiled code. The calling sequences are described fully in the sections following.

Oper.	Parameters	Description
SECALL	SCA, CLD, UDA, RET	Simple single entry call.
SFCALL	SCA, CLD, UDA, EFI, RET	Simple family entry call.
TECALL	SCA, CLD, UDA, DELAY, RET	Timed single entry call.
TFCALL	SCA, CLD, UDA, EFI, DELAY, RET	Timed family entry call.
TACCEPT	ENT, UDA	Simple accept statement.
SELCLR	SSI	Get caller (selective wait).
ENDRND	ASP, CLR, RET	End of rendezvous.

The registers used to pass parameters to these operations are as follows:

UDA	Address of an entry's Unit Data Area.
RET	A return address.
SCA	Address of the Subprogram Communication Area (see Unit Execution Support).
CLR	Address of the calling task's Task Control Block.
CLD	Address of the called task's Task Control Block.
ASP	Accepting (called) task's Saved Priority.
DELAY	Delay time, in clock ticks.

EFI	Index (member number) within an entry family.
ENT	Absolute entry index (there is a unique index for each entry and each family member within a task).
SSI	Address of a work area of type SelectRecord, used to accumulate information for a SELECTIVE WAIT statement.

3.3.5.3.1.7 Notation Conventions

The following sections describe the calling sequences which the compiler must produce in the compiled code. Most of these are descriptions of machine instructions. Portions which are described using Ada syntax (terminated by semicolons) indicate that normal code is generated for the Ada constructs used.

In addition to the types listed under Inputs, the following abbreviations are used:

n	An integer (greater than 0). For a SELECT statement, it specifies one of the alternatives.
T	A task name.
Cn	A conditional expression (a guard).
En	Name of a simple entry, or family member.
In	Absolute index of an entry or family member. The task's first entry has an index of 1, the second 2, etc. Each family member also has a unique index.
Dn	Delay time or time limit, in seconds.
Pn	A list of formal parameters.
Body_n	The sequence of statements in an ACCEPT body. The body is null if there are no statements between the DO and the END, or if the DO and the END are missing.
Code_n	Any other (possibly null) sequence of statements.
An	Address of a Unit Data Area or code for one of the alternatives of a SELECTIVE WAIT.
CF	Clock frequency, in ticks per second (1/SYSTEM.TICK).

SelectInfo A work area used by the operations which set up a SELECTIVE WAIT. It is of type SelectRecord.

Branch to X An unconditional branch to a code location labelled X (IBM 4341 "B" instruction).

Branch to X, Link R An unconditional branch to a code location labelled X, after saving the address following the branch instruction in register R (IBM 4341 "BAL" instruction).

3.3.5.3.1.8 Simple Entry Calls

Refer to Processing for a general description of rendezvous implementation.

The format of a simple entry call in Ada is as follows:

```
T.E1 (P1);  -- Call task T, entry E1, with parameter P1.
```

The compiler-generated code for an entry call is similar to that for a procedure call (see Unit Execution Support), but requires different register values, and branches to tasking operations rather than directly to a code prologue.

There are three types of entry calls: simple, timed, and conditional. For each of these types, the called entry may be a simple entry, or a member of an entry family.

In each of the six possible combinations, any non-register parameters must be copied to the Subprogram Communication Area, and the following registers must be set:

SCA = Address of the Subprogram Communication Area.
CLD = Address of the called task's Task Control Block.

In addition to SCA and CLD, a simple entry call to a single entry requires the following:

UDA = Address of the called entry's Unit Data Area.
Branch to SECALL, link RET (return address).

In addition to SCA and CLD, a simple entry call to an entry family member requires the following:

UDA = Address of the called entry family's Unit Data Area.
EFI = The index of the member within the family.
Branch to SFCALL, link RET (return address).

3.3.5.3.1.9 Timed Entry Calls

A timed entry call in Ada has the following format:

```
select
    T.E1 (P1);        -- Call task T, entry E1, with parameter P1.
    S1;               -- Code following the call (possibly NULL).
or
    delay D2;         -- Delay time D2 is in seconds.
    S2;               -- Code following delay (possibly NULL).
end select;
```

The compiled code for timed entry calls is similar to that for simple entry calls (see the preceding section).

In addition to SCA and CLD, a timed call to a single entry requires the following:

UDA = Address of the called entry's Unit Data Area.
DELAY = Delay amount, in clock ticks.
Branch to TECALL, link RET (return address).

In addition to SCA and CLD, a timed call to an entry family member requires the following:

UDA = Address of the called entry's Unit Data Area.
EFI = The index of the member within the family.
DELAY = Delay amount, in clock ticks.
Branch to TFCALL, link RET (return address).

TECALL and TFCALL each return a function value in the normal way (see Unit Execution Support). The return value is BOOLEAN TRUE if the entry was successfully called, or FALSE if the delay time expired. If code for S1 or S2 is present, the code following the call to TECALL or TFCALL must test this value, and execute the code for S1 or S2 accordingly.

3.3.5.3.1.10 Timed Entry Calls

A conditional entry call in Ada has the following format:

```
select
    T.E1 (P1);          -- Call task T, entry E1, with parameter P1.
    S1;                 -- Code following the call (possibly NULL).
else
    S2;                 -- Executed if no acceptor (possibly NULL).
end select;
```

The compiled code for conditional entry calls is identical to that for timed entry calls (see the preceding section), except that the value for DELAY must be 0 in both cases. TECALL and TFCALL will return TRUE if the entry was called, or FALSE if the ELSE is to be taken.

3.3.5.3.1.11 Simple Accept Statements

Refer to Processing for a general description of rendezvous implementation.

Code for an ACCEPT body resembles that for a procedure body (see Unit Execution Support), in that it has a Unit Data Area and code. However, there is no prologue, and at the end of the ACCEPT body's code is a Branch-and-Link to ENDRND, rather than a return to the caller.

The simple ACCEPT statement may be in two possible forms:

```
accept E1 (P1);                 -- Null ACCEPT body.


accept E1 (P1) do
    Body_1;                     -- Code of ACCEPT body.
end E1;
```

The generated code for a simple ACCEPT statement is as follows:

```
        ENT = Absolute index of the entry or family member.
        UDA = A1 (Address of the ACCEPT body's Unit Data Area).
        Branch to TACCEPT.

    A1: <Unit Data Area for ACCEPT body>
        Body_1;                             -- May be null
        Branch to ENDRND, link RET (return address).

        <Code following ACCEPT body>
```

3.3.5.3.1.12 Selective Wait Statement

The following is an example of Ada source code for a selective wait with only ACCEPT alternatives, and the corresponding generated code:

Ada Source

```
select
    when C1 =>
        accept E1 (P1) do
            Body_1;
        end;
        Code_1;
or
    when C2 =>
        accept E2 (P2) do
            Body_2;
        end;
        Code_2;
or
    when C3 =>
        accept E3 (P3) do
            Body_3;
        end;
        Code_3;

end select;
```

Generated Code

```
InitSelect (SelectInfo);

if C1 then
    SetOpen (I1, A1, SelectInfo);
    endif;

if C2 then
    SetOpen (I2, A2, SelectInfo);
    endif;

if C3 then
    SetOpen (I3, A3, SelectInfo);
    endif;

    SSI = Address (SelectInfo)
    Branch to SELCLR

A1: <UDA for Alternative 1>
    Body_1;
    Branch to ENDRND, Link RET
    Code_1;
    Branch to ES

A2: <UDA for Alternative 2>
    Body_2;
    Branch to ENDRND, Link RET
    Code_2;
    Branch to ES

A3: <UDA for Alternative 3>
    Body_3;
    Branch to ENDRND, Link RET
    Code_3;

ES: <End of Select Statement>
```

3.3.5.3.1.13 Selective Wait with Delay Alternatives

The following is an example of Ada source code for a selective wait with DELAY alternatives, and the corresponding generated code:

Ada Source

```
select
    when C1 =>
        accept E1 (P1) do
            Body_1;
        end;
        Code_1;
or
    when C2 =>
        delay D2;
        Code_2;
or
    when C3 =>
        delay D3;
        Code_3;
or
    when C4 =>
        accept E4 (P4) do
            Body_4;
        end;
        Code_4;
end select;
```

Generated Code

```
InitSelect (SelectInfo);

if C1 then
    SetOpen (I1, A1, SelectInfo);
    endif;

if C2 then
    SetDelay (D2*CF,A2,SelectInfo);
    endif;

if C3 then
    SetDelay (D3*CF,A3,SelectInfo);
    endif;

if C4 then
    SetOpen (I4, A4, SelectInfo);
    endif;

    SSI = Address (SelectInfo)
    Branch to SELCLR

A1: <UDA for Alternative 1>
    Body_3;
    Branch to ENDRND, Link RET
    Code_3;
    Branch to ES

A2: Code_2;
    Branch to ES

A3: Code_3;
    Branch to ES

A4: <UDA for Alternative 4>
    Body_4;
    Branch to ENDRND, Link RET
    Code_4;

ES: <End of Select Statement>
```

3.3.5.3.1.14 Selective Wait with Else

The following is an example of Ada source code for a selective wait with an ELSE case, and the corresponding generated code:

Ada Source

```
select
    when C1 =>
        accept E1 (P1) do
            Body_1;
        end;
        Code_1;
or
    when C2 =>
        accept E2 (P2) do
            Body_2;
        end;
        Code_2;
else

    Code_3;

end select;
```

Generated Code

```
InitSelect (SelectInfo);

if C1 then
    SetOpen (I1, A1, SelectInfo);
    endif;

if C2 then
    SetOpen (I2, A2, SelectInfo);
    endif;

SetDelay (0, A3, SelectInfo);
    -- Zero delay = ELSE

    SSI = Address (SelectInfo)
    Branch to SELCLR

A1: <UDA for Alternative 1>
    Body_1;
    Branch to ENDRND, Link RET
    Code_1;
    Branch to ES

A2: <UDA for Alternative 2>
    Body_2;
    Branch to ENDRND, Link RET
    Code_2;
    Branch to ES

A3: Code_3;

ES: <End of Select Statement>
```

3.3.5.3.1.15 Selective Wait with Terminate Alternative

The following is an example of Ada source code for a selective wait with a TERMINATE alternative, and the corresponding generated code:

Ada Source

```
select
    when C1 =>
        accept E1 (P1) do
            Body_1;
        end;
        Code_1;
or
    when C2 =>
        accept E2 (P2) do
            Body_2;
        end;
        Code_2;
or
    when C3 =>
        terminate;

end select;
```

Generated Code

```
InitSelect (SelectInfo);

if C1 then
    SetOpen (I1, A1, SelectInfo);
    endif;

if C2 then
    SetOpen (I2, A2, SelectInfo);
    endif;

if C3 then
    SetTerminate;
    endif;

    SSI = Address (SelectInfo)
    Branch to SELCLR

A1: <UDA for Alternative 1>
    Body_1;
    Branch to ENDRND, Link RET
    Code_1;
    Branch to ES

A2: <UDA for Alternative 2>
    Body_2;
    Branch to ENDRND, Link RET
    Code_2;
    Branch to ES

ES: <End of Select Statement>
```

3.3.5.3.1.16 Tasking Initialization Operations

The operations listed below perform all scheduler, task, and task list initialization.

```
procedure StartScheduler  -- Initialize AdaTasking package.
    ;

procedure InitList (      -- Initialize a dependency list.
    Dependents: in TaskListPtr
    );

function CreateTask (   -- Create a task on a dependency list.
    List:       TaskListPtr;     -- Activation list.
    Master:     TaskListPtr;     -- Dependency list.
    Code:       UDAptr;          -- Task body UDA location.
    Descriptor: TaskTypePtr;
    Priority:   PriorityNumber
    )
    return TCBptr;               -- Returns pointer to new TCB.

procedure ActivateTasks (  -- Activate all tasks on a list.
    List:       in TaskListPtr          -- List to activate.
    );
    --| Each block, subprogram, task, and library package with
    --| directly dependent tasks must call ActivateList as its
    --| first executable statement (immediately after
    --| establishing the exception handlers for the block).  In
    --| addition, ActivateList must be called for each list of
    --| tasks created by an allocator.
```

3.3.5.3.1.17 Task Termination Operations

The tasking operations in this group implement task termination and abort.

```
procedure TaskWait (             -- Wait for dependents to terminate.
    Dependents: in TaskListPtr   -- List of dependents.
    );
    --| Every block, subprogram, and task with directly
    --| dependent tasks must call TaskWait as its last
    --| executable statement.

procedure AbortTask (            -- Ada ABORT statement.
    Task:       in TCBptr        -- Task to be aborted.
    );
```

3.3.5.3.1.18 Simple Delay Operation

The tasking operation in this section handles delays. This operation is called via the normal subprogram calling sequence (see Unit Execution Support).

```
procedure TaskDelay (              -- Ada DELAY statement.
    Amount:     in Duration        -- Delay in clock ticks.
    );
```

3.3.5.3.1.19 Interrupt Operations

The operations described in this section control the association of interrupts with task entries.

```
procedure CatchInterrupt (         -- Declare interrupt handler.
    Task:       in TCBptr;                 -- Task to handle it.
    Entry:      in EntryIndex;             -- Entry within task.
    Interrupt:  in InterruptType;          -- The interrupt.
    );
    -- Called for each interrupt address clause,
    -- at the activation of the task.

function PseudoInterrupt (         -- Simulate an interrupt.
    Interrupt:  InterruptType;
    )
    return BOOLEAN;                -- FALSE if the interrupt
                                   -- is not being handled.
```

3.3.5.3.1.20 Task and Entry Attributes

These tasking operations provide values for task and entry attributes, with the exception of SIZE, STORAGE_SIZE, and ADDRESS.

```
function IsCallable (    -- Attribute T'CALLABLE.
    Task:       TCBptr
    )
    return BOOLEAN;
    --| TRUE if the task exists and is neither completed,
    --| terminated, nor abnormal.

function IsTerminated ( -- Attribute T'TERMINATED.
    Task:       TCBptr
    )
    return BOOLEAN;      -- TRUE if the task is terminated.
```

```
function CallerCount (  -- Attribute E'COUNT.
    Task:       TCBptr;
    Entry:      EntryIndex;
    )
    return Count;       -- Number of callers on entry.
```

3.3.5.3.1.21 Debug Support Operations

The following is a partial list of operations which will be provided to support the debugger.

```
procedure Suspend (     -- Suspend low priority tasks.
    Limit:      in PriorityNumber
    );
    --| Causes the scheduler to ignore tasks with
    --| priority less than or equal to the limit.

procedure Resume        -- Cancel Suspend.
    ;
    --| Causes the scheduler to consider all runnable
    --| tasks.
```

3.3.5.3.1.22 Other Tasking Support Operations

```
procedure GiveUpProcessor  -- Give control to scheduler.
    ;

function MyTCBP         -- Return TCBptr of running task.
    return TCBptr;
```

3.3.5.3.2 Processing

3.3.5.3.2.1 Queues and Lists of Tasks

A globally accessible variable contains a pointer to the current running task TCB.

The running task, tasks waiting for a caller (not delayed), calling tasks in rendezvous, and tasks waiting for their dependents to terminate are on no queue (but are still on a dependency list). All other tasks are also on either an activation list, a runnable task queue, an entry call queue, or the delay queue.

3.3.5.3.2.2 Task Creation

For each task to be created, CreateTask allocates and initializes a TCB, and places it on the specified activation list. There are separate activation lists for the current master and for allocators.

3.3.5.3.2.3 Task Activation

At the end of each allocator, and after the BEGIN of a block, ActivateTasks is called to activate each task on the corresponding activation list, and to add them to the appropriate dependency list. If any tasks complete due to exceptions during activation, ActivateTasks raises TASKING_ERROR.

3.3.5.3.2.4 Task Termination

When a task terminates, or when it opens a TERMINATE alternative by calling SetTerminate, or when a master completes and calls TaskWait, the master's dependency list is checked. If the master is completed, and there are no non-terminable tasks, then all tasks on the dependency list and its sublists are terminated. Any task waiting for a list thus terminated is made runnable, so that it can leave its current block.

3.3.5.3.2.5 Call, Accept, and Rendezvous

In order to minimize scheduling overhead and stack frame allocation, at the beginning and end of a rendezvous, entry calls are made as much as possible like procedure calls. If the called entry is open when an entry call is made, the rendezvous is started immediately with no scheduling operations. At the end of the rendezvous, the higher priority task continues to execute;

the only required scheduling operation is to make the other task runnable.

The ACCEPT body is executed as if it were a procedure; i.e. the caller's primary and secondary stack are used to execute the ACCEPT body's code. However, in order to preserve the identity of the acceptor as the running task, the acceptor's Task Control Block is used. In addition, the acceptor's static link is used, so that scoping is correct within the ACCEPT body. The priority used within the rendezvous is the higher of the two.

At the end of the ACCEPT body is a branch (and link) to ENDRND, which ends the rendezvous. If the caller has higher priority, the acceptor is set runnable, to continue with the code following the ACCEPT body, and a RETURN is made to the caller, thus completing the entry call. If the acceptor has higher priority, the caller is set runnable, to continue by executing a RETURN, and the acceptor continues immediately with the code following the ACCEPT body.

Figures 3-14 and 3-15 summarize the task state transitions and queues for entry calls and accept's.

3.3.5.3.2.6 Interrupts

Interrupts are equivalent to simple entry calls, in that interrupts which occur while an interrupt entry is closed are queued. However, if the task with the interrupt entry is unactivated or terminated, the interrupts are ignored.

A call to CatchInterrupt creates an entry in an interrupt table: for each interrupt, the TCB address and the entry index of the interrupt entry are inserted.

When an interrupt occurs for which there is a table entry, a counter for that interrupt is incremented, and a call to the entry is made. Interrupts which occur while a previous interrupt is being serviced cause their counters to be incremented, but cause no additional entry call. After every return from an interrupt entry call, the counter is decremented. If it is non-zero, another entry call is made to service the pending interrupt, and so on until the counter goes to zero.

When a task with interrupt entries terminates, its entries are removed from the interrupt table. Further occurrences of these interrupts will then be ignored.

Figure 3-14, State Transitions (Caller):

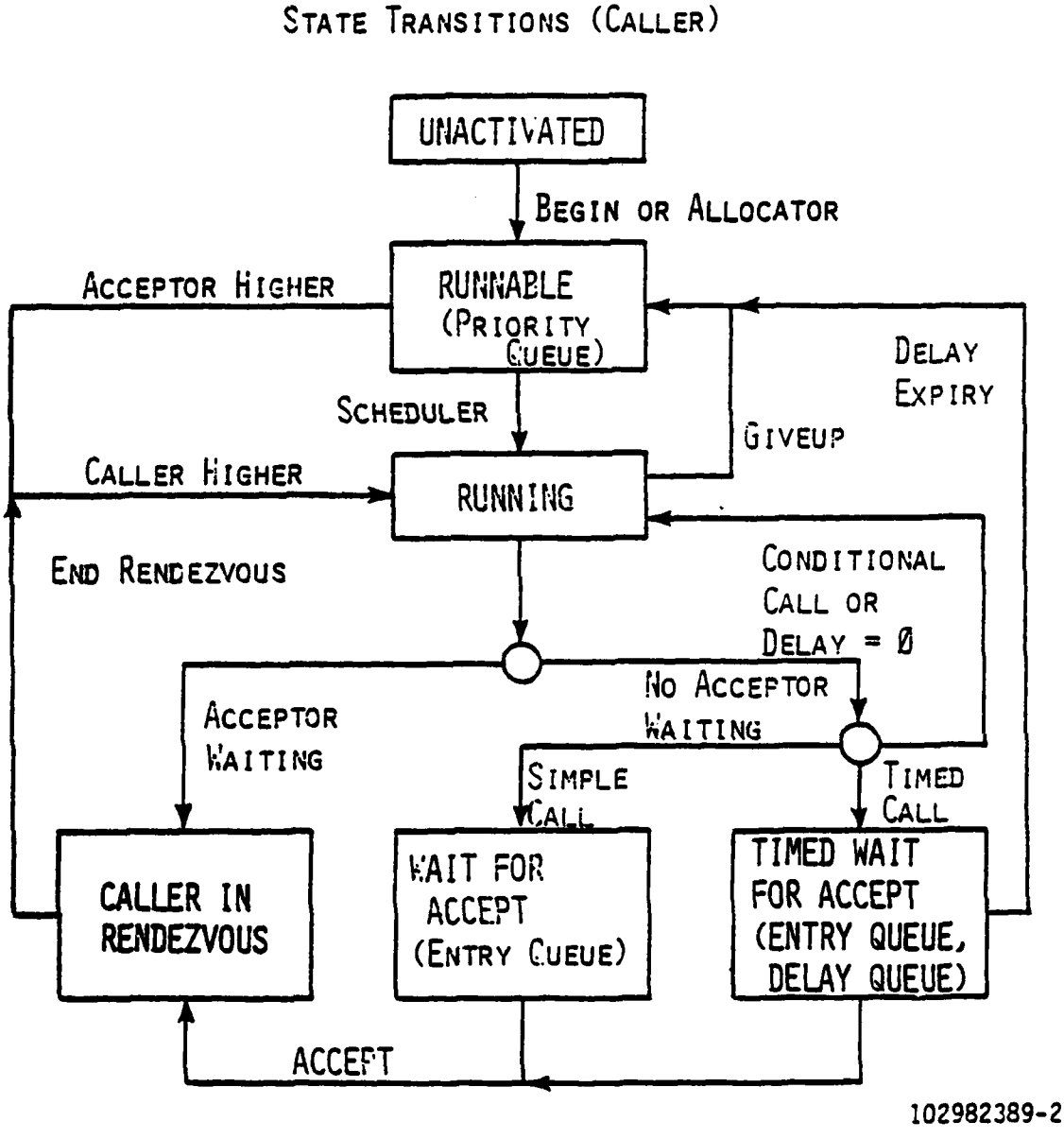
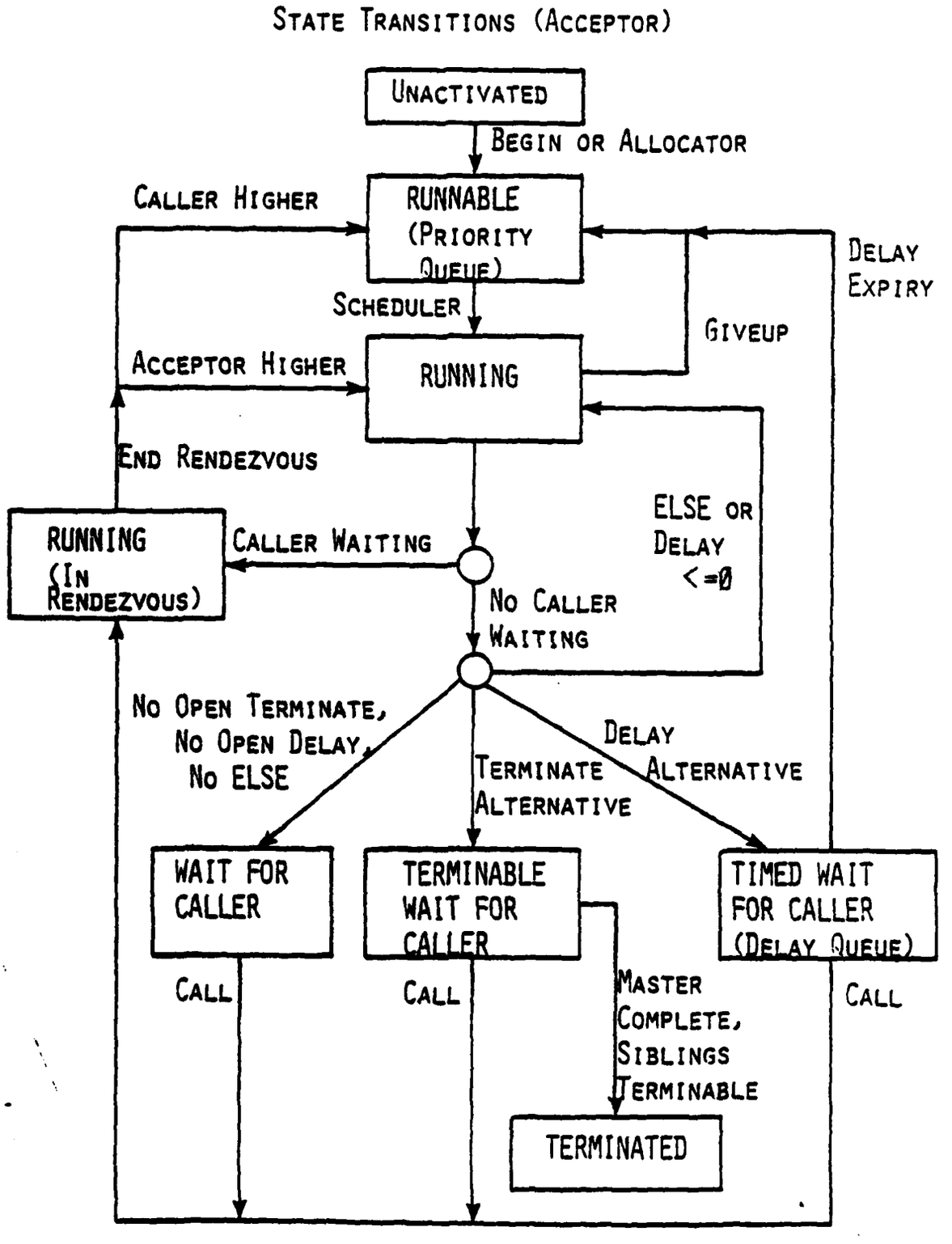


Figure 3-15, State Transitions (Acceptor):



102982389-1

3.3.5.3.3 Outputs

Nearly all AdaTasking operations produce changes in the state of the scheduler. Those listed below produce tangible outputs (see Inputs for a full description).

```
function  TECALL           -- Timed entry call.
function  TFCALL           -- Timed family member call.
procedure InitList         -- Initialize a task list.
procedure CreateTask       -- Create a task on a list.
procedure ActivateTasks    -- Activate tasks on a list.
function  PseudoInterrupt  -- Simulate a hardware interrupt.
function  IsCallable       -- Attribute T'CALLABLE.
function  IsTerminated     -- Attribute T'TERMINATED.
function  CallerCount      -- Attribute E'COUNT.
function  MyTCBP           -- Return TCB of running task.
```

3.3.5.3.4 Special Requirements

The following registers are set before entering an ACCEPT body, and must be preserved (or restored) at the end of the body:

ASP — Acceptor's Saved Priority.

CLR — Pointer to caller's Task Control Block.

Due to the special interfaces required, and to maximize the efficiency of rendezvous operations, the operations listed in Special Calling Sequences, which support entry calls, accept statements, and selective waits will be implemented in IBM 4341 machine language.

3.3.5.4 Exception Handling

The following sections discuss the implementation of the Ada exception handling mechanisms.

3.3.5.4.1 Inputs and Outputs

The exception handling package implements a variety of low-level operations which are invoked by the generated code.

3.3.5.4.1.1 Exception Identifiers

Each exception name declared within a program is assigned a unique 32 bit exception identifier. Exceptions in Ada are, in effect, statically declared; an exception declared in a recursive subprogram is the same exception for each recursive invocation.

3.3.5.4.1.2 Register Usage

The low-level operations use the following registers for passing parameters:

reg.	normal use
EXC	Current exception identifier.
LOC	Location at which exception was raised.
RET	Subprogram return address.
FRAME	Pointer to current call frame.

3.3.5.4.1.3 Exception Handling Operations

oper.	registers	description
RAISE	EXC, FRAME, RET, LOC	Raise exception.
PRCALLER	EXC, FRAME, LOC	Propagate exception to caller.
PRRENDV	EXC, FRAME, LOC	Propagate out of rendezvous.
EXABORT		Abort task due to unhandled exception.

3.3.5.4.1.4 Exception Maps

The compiler generates an exception map for for each executable subprogram, package, generic and task unit. A unit may have several handlers associated with it, since nested blocks and inline subprograms execute on the call frame belonging to the containing unit. A pointer to the map is contained in the unit data area (see Unit Execution Support) associated with the body. The map consists of a sequence of double-word entries, where the first word contains the address of the last instruction to which this particular entry applies, and the second word contains the address of the corresponding exception handling code. The entries are sorted in order of instruction address:

```
procedure A is
   ...                          -- elaboration code for procedure
   [instruction a]
begin
   ...
   declare
      ...                       -- elaboration code for nested block
      [b]
   begin
      ...
      [c]
   exception                    -- h1
      ...
   end;
   ...
   [d]
exception                       -- h2
   ...
   [e]
end A;
```

instruction	handler
a	PRCALLER
b	h2
c	h1
d	h2
e	PRCALLER

3.3.5.4.1.5 Hardware Detected Exceptions

The IBM 4341 hardware is capable of detecting a number of exceptions. The following hardware exceptions are intercepted by the run-time system and cause NUMERIC_ERROR to be raised:

exponent underflow
exponent overflow
floating point divide
significance
fixed point divide
fixed point overflow

All other hardware detected exceptions cause PROGRAM_ERROR to be raised.

3.3.5.4.2 Processing

3.3.5.4.2.1 Raising an Exception

When a raise statement with an exception name specified is seen by the compiler, code is generated to load the corresponding exception identifier into EXC, load the current program counter into LOC, and then branch to RAISE. A raise statement with no exception name differs only in that the exception identifier for the exception in progress is loaded into EXC. When a hardware detected exception occurs, the intercept routine loads the exception identifier for NUMERIC_ERROR or PROGRAM_ERROR into EXC, loads the address of the interrupted instruction into LOC, then branches to RAISE.

Upon entering RAISE, the current call frame (addressed by FRAME) is examined to obtain the address of the unit data area, which was saved in the call frame on entry to the unit. The address of the exception map is then obtained from the unit data area. If LOC currently points to an instruction within a run-time routine, then LOC is loaded with the address at which the run-time routine was called, which should be in RET. The exception map is then scanned for the first entry with an instruction address greater than LOC. When the entry is located, control is passed to the corresponding handler.

3.3.5.4.2.2 Exception Handlers

The code generated by the compiler for a set of exception handlers first performs a series of tests to determine if EXC specifies an exception for which the user has provided an explicit handler. If so, the corresponding code is executed. If not, and a handler was provided for the others choice, that code is executed. If the user did not provide an others choice, the compiler generates a branch back to the RAISE routine, causing the exception to be propagated to the enclosing exception frame; the contents of EXC and LOC are not disturbed.

When execution of a handler is completed without propagating an exception to a containing exception frame, control is passed to the normal exit code for that block or unit. When an exception is to be propagated out of the handler, the compiler generates code to, when needed, perform a dependent task wait, release secondary stack storage, and/or release the current primary stack frame, prior to branching to RAISE. If the user has not provided an exception handler for a block or inline subprogram which has dependent tasks, the compiler generates a handler which performs the dependent task wait, then branches to RAISE.

3.3.5.4.2.3 Propagating Out of a Called Subprogram

An entry in an exception map specifying PRCALLER as the handler is used to cause propagation of an exception out of a called subprogram. PRCALLER loads the contents of the return address field in the current call frame into LOC, stores the address of RAISE in the return address field, obtains the address of the normal exit code for the subprogram from the unit data area, then branches to that address. The exit code will, when needed, perform a dependent task wait and/or release secondary stack storage, restore the caller's registers, then pass control to the address indicated in return address field, which is now RAISE.

3.3.5.4.2.4 Propagating Out of a Rendezvous

An entry in an exception map specifying PRRENDV as the handler is used to cause propagation of an exception out of a rendezvous. Such an entry is always provided for the last instruction in an accept body. PRRENDV calls a tasking support routine to propagate the exception to the participating tasks.

3.3.5.4.2.5 Unhandled Exceptions

An entry in an exception map specifying EXABORT as the handler is used to force completion of a task which has an unhandled exception. Such an entry is always provided for the last instruction in a task body. EXABORT obtains the address of the normal exit code for the task from the task's unit data area, then passes control to it.

If an exception is propagated out of the main subprogram, a post-mortem routine is called which will display the information concerning the state of the program at the point at which the exception was raised, then pass control to the debugger (when present).

3.3.5.4.3 Special Requirements

Due to the special interfaces required and to achieve a reasonable level of efficiency, all storage management operations will be implemented in IBM 4341 machine language.

3.3.5.5 Language-defined Packages

3.3.5.5.1 Standard Input/Output Packages

3.3.5.5.1.1 Inputs and Outputs

The KAPSE will implement the complete set of Ada Language Standard Input/Output packages. The specifications given in chapter 14 of the Ada Language Reference Manual will serve as the official definition of these standard input/output Packages. The headers of the specifications of these packages are given here for reference:

```
package IO_EXCEPTIONS is
        NAME_ERROR      : exception;
        USE_ERROR       : exception;
        STATUS_ERROR    : exception;
        MODE_ERROR      : exception;
        DEVICE_ERROR    : exception;
        END_ERROR       : exception;
        DATA_ERROR      : exception;
        LAYOUT_ERROR    : exception;
end IO_EXCEPTIONS;


with IO_EXCEPTIONS;
generic
    type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
        type FILE_TYPE is limited private;
        type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
        type COUNT is range 0 .. Implementation_Defined;

    . . .
end SEQUENTIAL_IO;


with IO_EXCEPTIONS;
generic
    type ELEMENT_TYPE is private;
package DIRECT_IO is
        type FILE_TYPE is limited private;
        type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
        type COUNT is range 0 .. Implementation_Defined;

    . . .
end DIRECT_IO;
```

```
with IO_EXCEPTIONS;
package TEXT_IO is
        type FILE_TYPE is limited private;
        type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
        type COUNT is range 0 .. Implementation_Defined;
        subtype FIELD is INTEGER range 0 .. Implementation_Defined;
        subtype NUMBER_BASE is INTEGER range 2 .. 16;

    . . .
        generic
            type NUM is range <>;
        package INTEGER_IO is
            DEFAULT_WIDTH   : FIELD := NUM'WIDTH;
            DEFAULT_BASE    : NUMBER_BASE := 10;
            . . .
        end INTEGER_IO;
    . . .
end TEXT_IO;


package LOW_LEVEL_IO is
    -- declarations of the possible types for DEVICE and DATA;
    -- declarations of the overloaded procedures for these types:
    procedure SEND_CONTROL (    DEVICE: device type;
                                DATA: in out data type);
    procedure RECEIVE_CONTROL ( DEVICE: device type;
                                DATA: in out data type);
end;
```

3.3.5.5.1.2 Processing

Internally, all operations are converted to operations on storage unit arrays, allowing arbitrary types of objects to be handled. The conversion to standard types is made within the generic body of the package, while the bulk of the processing is done in a non-generic package to avoid multiple instantiations.

3.3.5.5.1.3 Special Requirements

3.3.5.5.2 Calendar Package

3.3.5.5.2.1 Inputs and Outputs

The KAPSE will implement the complete Ada Language Standard Calendar package. The specifications given in chapter 9 of the Ada Language Reference Manual will serve as the official definition of this package. The header of the specification of

this package is given here for reference:

```
package CALENDAR is
        type TIME is private;
        subtype YEAR_NUMBER     is INTEGER range 1901 .. 2099;
        subtype MONTH_NUMBER    is INTEGER range 1 .. 12;
        subtype DAY_NUMBER      is INTEGER range 1 .. 31;
        . . .
    end CALENDAR;
```

3.3.5.5.2.2 Processing

All CALENDAR operation will be implemented using well-known algorithms, such as described in D. E. Knuth, The Art of Computer Programming, Vol. 1: FUNDAMENTAL ALGORITHMS, Addison Wesley, 1975, pp 155 - 156. The basic unit for conversions will be type DURATION, which will be one clock tick (SYSTEM.TICK).

3.3.5.5.2.3 Special Requirements

3.3.5.6 Type Support Routines

Type support consists of a set of subroutines used by compiler generated code to do simple operations on typed objects. Type support run-time routines include: arithmetic on reals, image and value functions for discrete types, and POS, VAL, SUCC, and PRED functions for enumeration types with representation specs. These should not be confused with compiler generated routines for doing operations on objects of a particular type, such as array indexing routines.

3.3.5.6.1 Inputs and Outputs

Type support routines take as input and return as output scalar values, pointers to run-time system data structures that describe the mappings needed to convert position number to enumeration values, and position number to the string image of an enumeration value.

The arithmetic subprograms on reals include: add, subtract, negate, multiply, divide, and power, for the built-in types float, and long float as well as the built-in fixed point types. There are 32 built-in fixed point types, one for each possible position of the binary point in a 32-bit machine word. There is one set of arithmetic run-time subprograms for all fixed point

types; these routines take, in addition to their normal arguments, the position of the binary point of each argument.

There are IMAGE, VALUE, and 'WIDTH functions for each of the built-in numeric types: SHORT_INTEGER, INTEGER, FLOAT, and LONG_FLOAT, as well as one pair for the built-in fixed point types. Additionally, there is a set of conversion routines; there is one member in this set for each pair of built-in numeric types.

There are IMAGE, VALUE, and 'WIDTH functions for natural machine storage unit size of enumeration types, byte, half_word, and word. These routines take, in addition to the object or string passed, a pointer to a data structure, generated by the compiler, which maybe used to guide the mapping between string representation to position number.

There is a set of routines - POS, VAL, SUCC, and PRED - for each natural machine storage unit. Byte, half_word, and word, are provided for manipulating enumeration types that have received representation specfications. Like the IMAGE and VALUE functions for an enumeration type, these functions take an additional parameter: a pointer to a data structure generated by the compiler, that can guide the mapping between position number and a the user-specfied representation number.

3.3.5.6.2 Processing

The conversion and numeric operations are implemented in the usual manner. Where it is appropriate, the code for them is inserted inline by the back end during code generation. On a machine with hardware support for such functions the operations are generated by the compiler and no use is made of the run-time system routines.

The enumeration mapping functions, for conversion to and from strings, and for conversion to and from position number, use maps generated by the compiler for each particular enumeration type. Consider a possible string to position number map built for this enumeration type:

```
type color is (red,white,blue);
```

This map might be an instance of the following data structure:

```
type literal_image is record
        image_offset:   integer;
        image_width:    integer;
end literal_image;

type literal_images is array (integer range <>) of literal_image;
```

```
type enumeration_image_map (
        string_space:   integer,
        enum_length:    integer
        )
    is record
        map:    literal_images (1..enum_length);
        images: string (1..string_space);
end enumeration_image_map;
```

The particular map for this enumeration would then be the data structure generated by the compiler for this aggregate:

```
color_image_vs_pos_map: enumeration_image_map :=
        (
        string_space => 11,
        enum_length => 3,
        pairs_map => ((1,3), (4,5), (9,4)),     -- (posn,width)
        images => "redwhiteblue"
        );
```

The STORAGE phase of the compiler generates this "enumeration_image_map," for each enumeration type. The actual implementation may differ in detail from the one described here.

3.3.5.6.3 Special Requirements

3.4 Adaptation and Rehosting

3.4.1 Installation parameters

The following parameters must be supplied as part of installing a KAPSE on a particular host:

1. The block size;
2. The number of block buffers in the buffer cache;
3. The maximum number of simultaneous users;
4. The maximum number of simultaneous programs.

3.4.2 Operation parameters

The following parameters may be adjusted on a running KAPSE to reflect a changing operational environment:

1. The maximum memory allocation per program;
2. The limit on number of simultaneous programs per user;
3. Host-dependent scheduling parameters;
4. The names and numbers of device objects (see ***).
5. Processing and disk budgets (see ***).

3.4.3 Rehosting Requirements

Rehosting the KAPSE will require retargeting the Ada compiler and re-implemting the KAPSE/Host interface. The KAPSE/Host interface has been kept as simple and low-level as possible to facilitate rehosting to a new host system or bare machine.

Any host must provide some kind of direct access disk or other on-line storage device. The host must also provide some kind of asynchronous pseudo interrupt to implement Ada real-time constructs and inter-program communication.

3.5 Capacity

KAPSE performance will vary according to user load and host system speed and capacity. In addition to the above installation and operation parameters, the following parameters will have a significant impact on throughput and response time:

1. The current number of simultaneous programs;
2. The amount of database access;
3. The locality of database access;
4. The amount of inter-program communication;
5. The number of simultaneous interactive users.

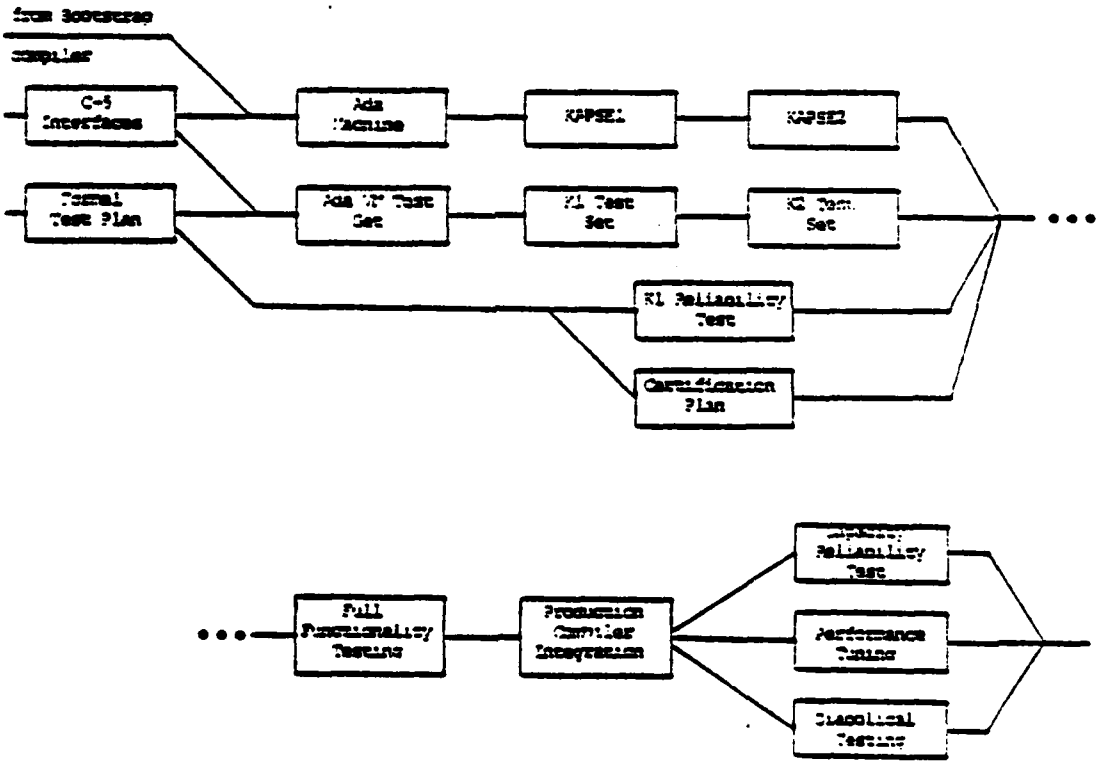
This page left blank intentionally.

4. QUALITY ASSURANCE PROVISIONS

4.1 Introduction

Because the KAPSE serves as the guardian of the entire database, the testing and validation procedure must be very intensive. The general approach is to use automation and parallel efforts to achieve a high level of confidence in a short time. These activities are illustrated below:

KAPSE Sub-Project



4.2 Test Requirements

4.2.1 Ada Machine Testing

The Run-Time System or "Ada machine" consists of implementations of all routines called implicitly by Ada programs, and the specifications and bodies of all subprograms defined by the standard environment. It includes storage management, tasking, exception handling, unit execution support, and type support routines. The Ada machine test consists of the ACVC compiler validation set.

4.2.2 Production Input/Output Tests

The next test/production phase covers basic database functions below the user level. These include the following functions:

a. KAPSE/Host interfaces;

b. Physical disk block allocation, reference counting, and read/write;

c. Logical disk block read/copy/write, with join-counting and automatic copy-on-write;

d. Data clumps and access methods, used to implement primitive files;

e. Primary windows on extended attributed objects;

f. User-defined attributes;

g. Path names via distinguishing attributes, creating and deleting simple object components of composite objects;

h. Primitive program invocation facilities;

i. A primitive history attribute, logging all KAPSE calls.

When the units listed above have been tested individually, the project begins to develop the production software on the system developed so far, rather than the bootstrap environment. Database integrity is the responsibility of a juman software librarian, who does manual backups daily. "Self-use," or further development of the KAPSE on the KAPSE is the primary form of integration testing at this point.

4.2.3 KAPSE Version 1 Test Case Generation

The scripts saved during this phase, especially those which failed or caused a system crash, will become the primary set of regression tests. The MAPSE project manager will run the

regression and other tests and commit the entire KAPSE/MAPSE project to the use of "KAPSE-1" as a development system, after the following additional features have been developed:

a. Category-defined attributes;

b. All remaining operations on components of composite objects;

c. Partitions and secondary windows;

d. Access control via roles and access control attribute;

e. Automatic backup and recovery.

The combined set of unit, integration, and regression tests developed by this point are a proposed AIE validation set (PAVS). They are used as an acceptance test for new releases of the KAPSE to the rest of the AIE project. A program will be developed to automatically run test suites once, or repeatedly, and check for correct execution of all tests.

4.2.4 K1 Reliability Test

The PAVS tests will be run cyclicly on the version 1 KAPSE for two weeks without crashing. It is estimated that four weeks of calendar time will be needed to debug version 1 to the point of surviving two weeks. This reliability testing overlaps additional development work in the areas of:

a. Full history and archiving support;

b. Configuration management tools;

c. Full program invocation and control, including private objects;

d. Login/Logout processing with user budgets and accounting;

e. System operation and maintenance procedures;

f. Full terminal screen management software.

4.2.5 Full Function Testing

After incorporating any changes indicated by the outcome of K1 reliability testing, and the list of new developments above, KAPSE version two and test set K2 are developed. Set K2 includes K1, specific unit tests for the new features, scripts saved from all K1 crashes, and other tests which will be required for government acceptance. Testing and debugging are continued until all K2 tests have been passed. Next the KAPSE is recompiled with the production compiler, and set K2 is repeated. KAPSE version three consists of version two as recompiled and re-debugged with

respect to test set K2.

4.2.6 KAPSE Version 3 Testing

Version three testing will proceed as three parallel efforts: The first will be the capacity and reliability test, consisting of running the full K2 set continuously for two weeks with a database constantly growing in number of objects, users, categories, etc. At the same time, there will be diabolical testing, consisting of giving skilled programers specific instruction and motivation to find ways to defeat access controls, corrupt the database, etc. And finally, as programmers make corrections and performance improvements, they will perform development testing.

4.3 Acceptance Requirements

The acceptance test consists of the K2 set, the capacity and reliability test, the scripts generated during successful and unsuccessful diabolical tests, and throughput tests to measure performance against the level A requirements.

INTERMETRICS INCORPORATED • 733 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840